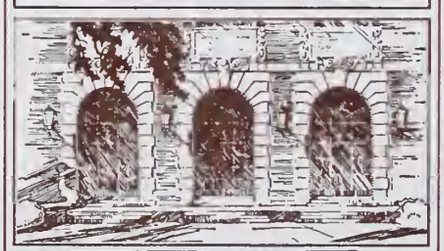


Math

8

Report No. UIUCDCS-R-76-832

AN AUTOMATIC VERIFIER FOR A CLASS OF SORTING PROGRAMS

by

PRABHAKER MATETI

October 1976

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-76-832

AN AUTOMATIC VERIFIER FOR A CLASS OF SORTING PROGRAMS

by

PRABHAKER MATETI

October 1976

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was supported in part by the National Science Foundation under Grant No. NSF EC 41511 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, October 1976.

ACKNOWLEDGMENTS

I am indebted to Jurg Nievergelt, my thesis advisor, for his encouragement, interest, help, advice and patience.

I am grateful to Jayadev Misra who had significantly influenced my thinking about programming; to Dave Plaisted for his help during the development of the theorem prover; to Dave Eland who always helped me out of TUTOR troubles; and to Ron Danielson without whose critical reading this thesis would have been even poorer in style.

I am thankful to Bert Speelpenning for a number of useful discussions, and to Wilfred Hansen for his encouragement and criticism.

Table of Contents

Table of Contents (Continued)

1. INTRODUCTION

This thesis discusses primarily the theoretical basis of a verifier for sorting programs designed for use in an automatic tutor for computer programming. A system, called SORTLAB, has been built embedding the sorting program verifier. SORTLAB allows a student to write programs for sorting an array, and decides whether these programs are correct; if they are not, it generates counterexamples. SORTLAB has been implemented on the PLATO system for computer-aided instruction [Alpert and Bitzer 1970], as a part of an Automated Computer Science Education System, ACSES [Nievergelt 1975].

The development of SORTLAB required several different components, in particular:

- a programming language convenient enough to write programs for sorting an array and not necessarily other programs,
- an assertion language with just the required expressive power to assert the state of an array with respect to the order of its elements,
- special purpose techniques for verifying these programs with assertions, including a theorem prover for the class of lemmas generated by the verifier.

1.1 Automatic Program Verification

With the increased concern for program reliability, the verification of programs is receiving greater attention than ever before. The verification process consists of checking if the program meets its

specifications, namely, that it always terminates, and when it does, certain variables have a desired property provided the input given to the program meets the input specification.

The inductive assertion method of proving programs considers these two problems separately: That the program meets its input/output specifications is proven separately from that of proving termination. The method also requires that an invariant property about the program variables be given for every loop. Given these specifications and loop assertions, a set of mathematical lemmas are generated, which depend on the assertions given, and the semantics of the programming language used. If these lemmas are true, the correctness of the program is guaranteed; thus proving that a program meets its specification is equivalent to proving a certain set of lemmas. This is the crux of the problem.

Much of the verification process is of a very mechanical nature, and unless a large part of the process is carried out by the computer, few programmers would be willing to hand-verify their programs. A number of program verifiers have been constructed (see survey by London [1972]), requiring varying degrees of human intervention. However, these are far from being helpful to a programmer for several reasons. Typically, human aid is required in pruning the proof-trees. An ordinary programmer is not trained in theorem-proving, and is usually not interested in <u>how</u> these lemmas are proved. In addition, if the program is incorrect, the verifiers cannot provide assistance either by generating a counter-example, or by pointing out where the error lies. Finally, the verifiers are slow in operation, even for small programs. These conditions

combine to tempt a programmer test-run his programs rather than submit them to an automatic verifier!

1.2 Limited Domain Program Verifiers

The failure in constructing verifiers that are mechanical aids to program writing can be attributed largely to the ambitious approach taken in building these verifiers. Except for the earliest of the verifiers [King 1969], the others have been increasingly ambitious in the variety of programs they intended to verify. The wide scope of programs being proven requires that the programs be written using elementary but powerful operations. Further, the assertions, and hence, the lemmas generated have to be formulated in first-order predicate calculus (or the equivalent), which is theoretically undecidable. By increasing the power of the theorem provers, we not only make them nondecision procedures, but they also lose a sense of direction toward their goal. A large number of useless inferences are then generated. Even among the decidable domains of problems, the theorem provers must be carefully designed in order to yield a decision procedure that works in practice. A "good" theorem prover should prove a large class of theorems that are often encountered very quickly, while it may take a while to decide about others.

A verifier and its theorem prover can become simple, if they incorporate certain aspects of the semantics of the problem domain. For example, most well-written nonnumeric programs manipulate their data structures in a "disciplined and uniform" way, which is as yet not formally characterizable. The verification lemmas arising from such

programs seem to be of a different nature from those that may arise in ordinary mathematics, say, number theory. If this is indeed the case, the underlying formal system may be decidable. If so, an incorrect program may be proven to be incorrect, counterexample generation may be feasible, and fast theorem-proving procedures may exist for the specific class of lemmas.

Strictly speaking, every program verifier constructed so far is a limited domain verifier. For example, the programs being verified are often limited to those that operate on integer-valued variables. But we mean to limit the domain even further. Some examples of such domains are programs operating on linear arrays with no arithmetic, those using lists, binary trees, etc.

It is doubtful if it would ever be possible to construct successful general purpose program verifiers. On the other hand, practical verifiers dealing with programs from a limited domain of discourse can be designed. This thesis provides one such example, namely, a verifier for in-place sorting programs which is being used in an automatic tutor of computer programming.

1.3 Program Verification in Teaching Programming

It is important that a student programmer realize the need for program reliability. A concern for the correctness of programs at an early stage in one's education has great impact on one's attitudes toward programming in later years. As exemplified by Dijkstra and others, a systematic method of designing abstract programs depends heavily on

the correctness proofs of programs. The "elegance" of a program is usually directly proportional to the ease with which it can be proven correct. There can be no question that one's understanding of one's own program is increased greatly after inventing the loop assertions for the program. Quite often one discovers better ways of writing the program.

In teaching programming, one would like to supervise the program design process by the student, as well as examine thoroughly the finished product. Both these aspects are amenable to computerization, particularly if an interactive computer system is available. The teacher-program supervising the program design process should be an expert in the programming problem domain, must have an "opinion" about various design methodologies, and, perhaps more importantly, be able to converse with the student in a reasonable language. If the problem domain is sufficiently simple, such teaching programs can indeed be designed. For an example of such a system, see [Danielson 1975].

On the other hand, a teacher program examining the student's finished program will not, and should not, consider the design process. Regardless of how it was constructed, judging the program's correctness and elegance should be its concern. The teacher program may, at one extreme, simply test run the student program, or, at the other extreme, attempt to formally verify the student program. Such teacher program should contain at least a program editor, a run-time system, a program verifier, and a counterexample generator.

Apart from these technical qualifications required of the teacher programs, they should be fast enough to give interactive response

to the student. These considerations lead us to write a specialized teacher program with built-in knowledge of a programming domain resulting in an interactive programming laboratory, SORTLAB, wherein the student can prepare a sorting program, and use the program verifier iteratively until a correct program is obtained.

1.4 The Sorting Program Verifier

We have chosen in-place sorting as the limited domain of discourse in SORTLAB because of two main reasons. First, every program verifier constructed so far has verified several sorting programs; their authors quote, quite often exclusively, these examples. This gives us a basis for comparison. Secondly, sorting programs are perhaps the most used examples in introductory programming courses.

The verifier can actually prove any program, sorting or not, written in our mini-programming language and whose behavior can be asserted in the assertion language. (See Sections 2.2 and 3.1 for a description of these languages.) If the program is not proven correct, then there must be mathematical "lemma"(s) generated from the program and its assertions which are false. The verifier can generate counter-examples to these lemmas.

2. VERIFIER

Every program operates on a certain set of data objects and aims to produce an output set of data objects with desired properties. A subset of these data objects, the input, is given to the program, and the remaining data objects are the result of program execution. Quite often, the input changes in its structure, data objects get created or destroyed, their structure and relationships change. The program is expected to realize a desired property on the output only if the input meets certain requirements. To this end, the programmer asserts what relationships are to hold on the input data objects, and what holds on the output.

2.1 Inductive Assertion Method of Verification

Given the input and output assertions, say ϕ and ψ, we are interested in verifying that the program P behaves properly, i.e., whenever P is given input satisfying ϕ, the output satisfies ψ, if and when P terminates. Notationally, following [Manna and Pnueli 1974], let us express this statement by:

$$\{\phi \mid P \mid \psi\} \qquad (2.1)$$

The program P is said to be partially correct with respect to ϕ and ψ if (2.1) is true. (Occasionally we refer to ϕ and ψ as the entry and exit assertions of P when P is a program segment.) P is said to be totally correct, if in addition to (2.1) being true, P always terminates. In this thesis we will be dealing with partial correctness only, and

hence forth refer to this simply as correctness. We shall use "prove a program segment" as an abbreviation of "prove a program segment correct with respect to its entry and exit assertions."

The proof of (2.1) is trivial, in theory, if the set of data objects satisfying the entry assertion ϕ is finite, since the program can be checked separately on each of these data objects. However, in general this set is infinite, and even if it is finite, it is usually such a large set that separate treatment of each data object is not practical.

One of the most widely used verification techniques, the inductive assertion method [Floyd 1967, Naur 1966], divides the verification process into two phases. First, a set of mathematical lemmas ("verification conditions") is generated, which, if proven, is sufficient to imply the correctness of the program. The second phase is the proof of the lemmas thus generated.

The generation of the verification conditions is intimately linked to the semantics of the programming language being used, and to the structure of the program at hand. The program being proven is decomposed into loop-free segments such that each segment has an entry assertion and an exit assertion. Any computation performed by the program is then a concatenation of executions of some selected segments. Thus, if each segment is correct with respect to its entry and exit assertions, using induction on the number of loop iterations, it can be proved that every computation with input satisfying the input assertion of the program yields output satisfying the output assertion when the program terminates.

Once the validity of this inductive proof is established, to prove a given program, only the decomposed loop-free segments need be proven. Requiring each loop to include an invariant assertion guarantees the decomposition of the program into loop-free segments each with entry and exit assertions.

Section 2.3 describes the generation of these lemmas. The forward substitution method is touched upon only briefly as we shall be using the backward substitution method. The programming and assertion languages used are described informally in Section 2.2; their formal specification appears in Chapter 5. Figures 2.1 and 5.2 give examples of programs written in our languages. A decision procedure for the lemmas is presented in Chapter 3.

2.2 The Programming and Assertion Languages

In the interest of developing a fast and small verifier, we limit the domain of programs. But the variety of programs cannot be limited by a programming language alone. As is well known [McCarthy 1960], a programming language rich enough to include a successor function, a conditional, and recursion is universal in the sense that any recursive function can be programmed in this language. A programming language, by imposing constraints and providing certain kinds of primitive operations while eliminating others, can only make it very inconvenient, but not impossible, to write certain programs.

On the other hand, an appropriately chosen assertion language can limit the kind of programs that can be asserted in that language. A

```
1  procedure sort  (n)
*  true
2    scan down with i from n to 2
3      scan up with j from 1 to i-1
4        if  xj > xj+1 then
5         exchange xj with xj+1
6        else
7        endif
*        1 ≤ J < I ≤ N & A(1;J) ≤ XJ+1 & A(1;I) ≤ S(I+1;N)
8      endscan
*      1 < I ≤ N & A(1;I-1) ≤ S(I;N)
9    endscan
*  S(1;N)
1Ø endproc
```

Abbreviations

A(	for	array(
S(	for	sorted(
array(s,t) ≤ sorted(u,v)	for	array(s,t) ≤ array(u,v) and sorted(u,v)

Figure 2.1 A Bubble Sort Program with Assertions

more general assertion language will use elementary, but powerful, atomic predicates. This use of elementary predicates makes it difficult to lump together all related predicates. The loss of power of expression in a limited assertion language is compensated for by the large and recognizable chunks of properties in the assertion. Further, while it has been advocated that theorem provers make large inferences, it appears necessary that related information should be recognizable as such before large inferences can be made.

These considerations led us to design a mini-programming language and an assertion language which are specific to the sorting of arrays. Formal specification of the language is given in Chapter 5. Below we touch upon only the salient features.

2.2.1 Programming Language

Operations on Keys

It is a well-recognized principle in program design that basic procedures, specific to the particular problem and the data structures being used, should be developed and used so that data integrity may be preserved [Dahl et al. 1972]. Sorting programs must conserve the keys they are sorting. Hence, we provide two basic operations: exchange and insertion of keys, and forbid value assignments to the keys of the array. This guarantees that the elements of the array are conserved throughout the program. Therefore, our verifier need only prove that the array is sorted.

Operations on Array Indices

Successor and predecessor functions on the indices ("ptrs") of the array provide sequential access to the elements. A ptr variable may be assigned the value of a ptr expression, which is of the form <ptr variable> + <integer constant>.

Procedures

In our verification system, we assume that the "intention" of any procedure is to produce a certain permutation of the elements of the array x, which is global to all procedures. Most procedures, however, permute only the elements belonging to a certain contiguous segment, say array (a,b); thus, we require that each procedure have exactly two input (formal) ptr variables a, b. All elements not belonging to array (a,b) are made "read only" to this procedure; no such element is permitted to participate in any exchange or insert operation. "Guard expressions" are provided for this (see also [Marmier 1975], p. 57).

For simplicity, we insist that the formal ptr variables a, b be not subject to assignment (i.e., they may not appear on the left hand side of any assignment). The two variables must, of course, be distinct. Optionally, a procedure may return ptr results to the calling procedure. There are no global ptr variables. An entry and an exit assertion for the body of the procedure must be given.

Procedure Calls

A procedure call must contain two ptr expressions as the actual

input parameters for the procedure called. If the called procedure has output parameters, the call must receive these results in distinct ptr variables. We further insist that these variables be distinct from those appearing in the input ptr expressions; this is done for the sake of simplicity.

For each call statement, we require that an entry assertion to the call be given. Thus, the user must give not only the loop invariants, entry and exit assertions for procedures, but also an entry assertion for each call (see Section 6.1.2 for a related discussion).

Control Structures

In addition to the familiar if and while statements, a scan statement (similar to the for statement of other languages) is provided. The loop variable of scan, however, is not considered "unmodifiable" by its body.

2.2.2 Assertion Language

Array Predicates

When sorting arrays with sequential access, we generally need atomic predicates to indicate that:

1. The array segment from index s to index t is sorted
   $\equiv$ if $s \leq i < j \leq t$ then $x_i \leq x_j$, and

2. Elements of the segment from s to 5 are all less than any element of the segment from u to v

$\equiv$ if $s \leq i \leq t$ and $u \leq j \leq v$ then $x_i \leq x_j$

where x is the name of the array, and an index is a ptr expression. These array predicates are abbreviated as:

sorted (s,t)

and

array (s,t) $\leq$ array (u,v)

respectively. The segments are defined by their lower boundaries s, u and the upper boundaries t, v.

Ptr Predicates

Predicates relating the indices of the array will also be needed:

ptr i is at least c units below j $\equiv$ i + c $\leq$ j

where i, j are ptr variables and c an integer constant, and i + c is a ptr expression.

Assertions

An assertion, then, is a sentence formed of these basic predicates and the logical connectives and and or. Notice the absence of negation in this language which makes it impossible to assert that an array is NOT sorted. However, ptr predicates, e.g., i + c $\leq$ j, can be negated as j + (1-c) $\leq$ i. Henceforth, the assertions will be written

informally; e.g., $i + c > j$ rather than $j + 1 - c \leq i$ or $i = j$ rather than $i + 0 \leq j$ and $j + 0 \leq i$.

2.3 Verification Condition Generator

We first discuss the generation of verification conditions of a simple loop program segment W with a loop-free body S.

```
W:  while B do
        S                                        (2.2)
    endwhile
```

This is then generalized to cover arbitrary procedures. Two general methods for the generation of verification conditions are forward substitution and backward substitution [King 1969].

2.3.1 Forward Substitution

Let ϕ_W be the entry assertion of W, and ϕ_S be the entry assertion of S. We then symbolically execute S on ϕ_S to obtain an assertion $Sf(\phi_S)$. Then the exit assertion of W is generated as

$$\phi_W \text{ and not } B \text{ or } Sf(\phi_S) \text{ and not } B \tag{2.3}$$

The lemmas to be proven are:

$$\phi_W \text{ and } B \text{ logically implies } \phi_S \tag{2.4}$$

$$Sf(\phi_S) \text{ and } B \text{ logically implies } \phi_S \tag{2.5}$$

Proving (2.4) and (2.5) guarantees that the entry assertion ϕ_S of S will be true each time S is entered.

The assertion $Sf(\phi_S)$ can be obtained by forward substitution as follows: If S is empty then $Sf(\phi_S)$ is the same as ϕ_S. Otherwise, let S be a concatenation of S1 and S2, where S2 may be empty. Then we obtain $Sf(\phi_S)$ by recursively applying rules F1 and F2 defined as follows:

Rule F1 (applicable iff S1 is an assignment statement)

Let S1 be $u \leftarrow t$ where t is an expression

then $Sf(\phi_S)$ is

S2f (subst u' for u in ϕ_S) and u = (subst u' for u in t) (2.6)

Rule F2 (applicable iff S1 is an if statement)

Let S1 be

```
if B1
then S3
else S4
endif
```

Then $Sf(\phi_S)$ is

S2f(S3f (W and B1) or S4f(W and not B1)) (2.7)

where subst y for z in F stands for the expression obtained by substituting y for all occurrences of z in the expression F. The variable u' refers to the previous value (before S1) of the variable u; thus (2.6)

asserts the existence of a value for u which satisfied ϕ_S prior to S1 which is to be used in the expression t. This introduction of existential quantifiers causes certain technical difficulties to our theorem prover (see Chapter 3). Hence we have chosen to abandon forward substitution, even though it seems appealing due to its close association with ordinary execution of programs, and to adopt the backward substitution method, which does not introduce any quantifiers.

2.3.2 Backward Substitution

Without loss of generality, let the given loop invariant be the exit assertion of the loop body S. Given the exit assertion ψ_S of S, we generate an entry assertion ϕ_S such that $\{\phi_S \mid S \mid \psi_S\}$. It should be noted that several such assertions ϕ_S exist, one of them being the trivial false. However, the ϕ_S generated by backward substitution is such that, for any ϕ', if $\{\phi' \mid S \mid \psi_S\}$ then ϕ' logically implies ϕ_S.

Now let ψ_W be the exit assertion of W, and ψ_S be the exit assertion of the loop body S. We can symbolically "unwind" the execution in the backward direction and obtain the entry assertion of S as $\phi_S \equiv Sb(\psi_S)$. Then the entry assertion of W is

$$\phi_W \equiv \phi_S \text{ and } B \text{ or } \psi_W \text{ and not } B \tag{2.8}$$

Proving the lemmas

$$\psi_S \text{ and } B \text{ logically implies } Sb(\psi_S) \tag{2.9}$$

and

$$\psi_S \text{ and not } B \text{ logically implies } \psi_W \tag{2.10}$$

guarantees that the entry assertion $\phi_S \equiv Sb(\psi_S)$ of S will be true each time S is entered, and that the exit assertion ψ_W of W holds, when the while-loop is exited.

The assertion $Sb(\psi_S)$ is obtained by backward substitution as follows: Let S be a concatenation of S1 and S2, S ≡ S1; S2 where S2 may be empty. We consider two cases.

S1 is not a call statement

We recursively apply rules B1 and B2 to obtain $Sb(\psi_S)$.

Rule B1 (applicable iff S1 is either a ptr-assignment, exchange, or insert statement)

B1.1: Let S1 be u ← t where t is a ptr expression

Then ϕ_S is

$$Sb(\psi_S) \equiv \underline{subst}\ t\ \underline{for}\ u\ \underline{in}\ S2b(\psi_S) \qquad (2.11)$$

B1.2: Let S1 be $\underline{exchange}\ x_a\ \underline{with}\ x_b$ where a, b are ptr expressions. Then,

$$Sb(\psi_S) \equiv \underline{exchb}\ x_a\ \underline{with}\ x_b\ \underline{in}\ S2b(\psi_S) \qquad (2.12)$$

B1.3: Let S1 be $\underline{insert}\ x_a\ \underline{below}\ x_b$ where a,b are ptr-expressions. Then,

$$Sb(\psi_S) \equiv \underline{nsrtb}\ x_a\ \underline{below}\ x_b\ \underline{in}\ S2b(\psi_S) \qquad (2.13)$$

We postpone (to Chapter 3) an accurate description of these inverse functions exchb . . ., and nsrtb. . ., as their evaluation plays an important

role in our theorem proving. Intuitively, these functions reproduce the situation(s) which must have existed prior to the exchange or insert.

Rule B2 (applicable iff S1 is an if-statement)

Let S1 ≡ if B1 then
S3
else
S4
endif

Then

$Sb(\psi_S)$ ≡ S3b(S2b(ψ_S)) and B1
or S4b(S2b(ψ_S)) and not B1

S1 is a call statement

Let the call statement and called procedure be as shown below:

* α	procedure Q(a,b) : (c,d)
call Q(s,t) : (u,v)	* ϕ_Q
* β	[procedure body]
	* ψ_Q
	endproc

where a and b are the two distinct input variables of procedure Q receiving values from the ptr expressions s and t respectively; the lists given after ":" are the output parameters; α is the given entry assertion to the call statement, and β is the generated exit assertion of call

looking at the statements below call; ϕ_Q, ψ_Q are the given entry and exit assertions for the procedure Q.

We should prove the following two lemmas:

$$\alpha \text{ logically implies } \phi_Q^s \tag{2.16}$$

$$\text{unmodified parts of } \alpha \text{ wrt } (s,t) \text{ and } \psi_Q^s \text{ logically implies } \beta \tag{2.17}$$

where ϕ_Q^s, ψ_Q^s, unmodified parts of α are obtained from ϕ_Q, ψ_Q, and α as described below.

The entry assertion ϕ_Q should not have any ptr variables other than a or b because these are not defined on entry. We substitute in ϕ_Q the expressions s and t for a and b, resulting in ϕ_Q^s.

The exit assertion ψ_Q should not have ptr variables other than a and b or those contained in ptr expressions c and d. We substitute in ψ_Q, s, t, u and v respectively for a, b, c and d to obtain ψ_Q^s. Note that the ptr variables u and v are substituted for expressions c and d. Also note that if c or d contains either a or b then the ptr variables u or v will be equal to an expression involving s or t. The substitution of s and t for a and b is valid because the variables a and b are not subject to assignment in procedure Q.

Recall that procedure Q can permute elements belonging to the segment array (a,b). Thus, those predicates of α not including segments which are strict subsegments of array (s,t) will still be true upon exit from Q. Such predicates are collected together in unmodified parts of α wrt (s,t). We postpone the description of this function to Section 3.3.2.

When the program consists of more than one procedure, we must prove the lemmas (2.16) and (2.17) for each procedure call, and further prove that the called procedures meet their specifications.

To generate the entry assertion for the body of a given procedure, we begin at the bottommost and innermost loop and successively generate the entry assertions of loops as described above. If ϕ_P is the generated entry assertion of the procedure body P, and ϕ is the given entry assertion, we should prove, in addition to the lemmas generated for each loop as in (2.9) and (2.10), the lemma

$$\phi \text{ logically implies } \phi_P \tag{2.15}$$

Clearly, the proof of all these lemmas guarantees that

$$\{\phi \mid P \mid \psi\} \tag{2.1}$$

An example of lemma generation appears in Figures 2.2 and 2.3.

```
1   procedure sort (n)
*   TRUE
2     i ← n
3     while i ≥ 2 do
4       j ← 1
5       while j ≤ i-1 do
6         if xj > xj+1 then
7           exchange xj with xj+1
8         else
9         endif
*         1 ≤ J < I ≤ N & A(1;J) ≤ XJ+1 & A(1;I) ≤ S(I+1;N)
10      endwhile
*       1 < I ≤ N & A(1;I-1) ≤ S(I;N)
11    endwhile
*   S(1;N)
12  endproc
```

The assertions 10* and 12* of this program are related to 7* and 8* of Figure 2.1 as follows:

$$10^* \equiv \underline{\text{subst}}\ j - 1\ \underline{\text{for}}\ j\ \underline{\text{in}}\ 7^*$$

$$12^* \equiv \underline{\text{subst}}\ i+1\ \underline{\text{for}}\ i\ \underline{\text{in}}\ 8^*$$

Figure 2.2 Bubble Sort Program of Figure 2.1 Rewritten using <u>while</u>-Statements Instead of <u>scan</u> s.

The verification conditions for the program in Figure 2.2 are:

- for loop at 5:

10* and j < i logically implies stmts[6 . .10]b(10*) (V1)

10* and j ≥ i logically implies subst i-1 for i in 12* (V2)

where

stmts[6. . .10]b(10*) ≡ the generated entry assertion of body of loop 5.

≡ $x_j \leq x_{j+1}$ and 9* or $x_j > x_{j+1}$ and exchb x_j with x_{j+1} in 9*

where

9* subst j+1 for j in 10*

- for loop at 3:

12* and i > 1 logically implies stmts[4. . .12]b(12*) (V3)

12* and i ≤ 1 logically implies sorted(1,n) (V4)

where

stmts[4. . .12]b(12*) ≡ subst 1 for j in
(subst i-1 for i in 12*) and j ≥ i
or stmts[6. . .10]b(10*) and j < i

- and for proc body:

true logically implies subst n for i in (V5)
sorted(1,n) and i ≤ 1
or stmts[4. . .12]b(12*) and i > 1

Figure 2.3 The Five Verification Conditions Generated for Bubble Sort

3. THEOREM PROVER

In this chapter, we describe a procedure for proving or disproving a theorem whose premise and conclusion are augmented well-formed formulas. Well-formed formulas (wffs) are the sentences of the assertion language (see Chapter 5). Augmented wffs involve the functions subst, exchb, and nsrtb which map a pair consisting of a wff and programming language statement into a wff. The details of these mappings will be given in Section 3.3. The proving or disproving of a theorem is done in two phases: in the first phase, the augmented wffs are converted into wffs; in the second phase, the actual proof begins. We discuss the second phase first, as the evaluation of the above functions involves the concept of partitioning which is an important part of the second phase.

The basic theorem prover is described in Section 3.1. A proof that this basic theorem prover is a decision procedure for theorems stated as wffs is given in Section 3.2. The theorem prover is then extended (in Section 3.3) to include evaluation of the aforementioned functions. The chapter concludes with Section 3.5 where the generation of counterexamples is discussed.

The treatment will be informal. The level of rigor in the proofs is comparable to that generally found when discussing combinatorial algorithms. Several "remarks" are made soon after describing a procedure. These are meta-lemmas describing the properties of the verification system. We omit the proofs of these remarks as they are neither illuminating nor interesting.

3.1 Basic Theorem Prover

A theorem prover attempts to prove that a given conclusion ω follows from a certain hypothesis or premise Ω. If ω does not follow from Ω, a general theorem prover may not always halt and say so. However, if Ω and ω are sentences of a properly chosen assertion language, it is possible, as we demonstrate in this chapter for our assertion language, to give a decision procedure for the question: Does Ω logically imply ω? We construct a "most general model" for Ω, and then determine if ω is "true" in this model. If ω is indeed satisfied by this model, then Ω logically implies ω; otherwise, we will be able to produce a counterexample which gives specific values to the variables making ω false and Ω true. To make the discussion more precise, we will need the following definitions.

3.1.1 Definition

Def 1 An interpretation of a wff is a mapping of the set of all ptrs, constants and the elements of the array into the set of integers. The ptr constants 0, 1, 2, . . ., and the function symbols +, -, relation symbols $\leq$, $<$, $>$, $\geq$, $=$, $\neq$ are given the usual meaning. The key function x maps the ptr expressions into key elements

We have taken the set of integers as the universe for the keys of the array only for the sake of simplicity in the ensuing discussion. However, any set of keys on which a linear ordering is defined will do for this universe. The domain of ptr values can similarly be enlarged.

Def 2 (Truth of Predicates):

The ptr predicates are interpreted as relations on integers in the conventional way.

The array predicate array (s,t) ≤ array (u,v) is true if either of the arrays is empty (i.e., s > t or u > v), or if no element of array (s,t) is greater than any element of array (u,v). (Similar meanings are given to <, >, and ≥ relations between array segments.)

The array predicate sorted (s,t) is true either if the array segment array (s,t) is empty, or if the elements of the segment are arranged in nondecreasing order from the lower boundary s to the upper boundary t of the segment.

A disjunct is of the form p_1 and p_2 and . . . where each p_i is a predicate. A wff is of the form d_1 or d_2 or . . . where each d_i is a disjunct. The logical connectives and and or are interpreted in the conventional way.

Def 3 An interpretation M is said to satisfy a wff ϕ, notationally $\models_M \phi$ if ϕ is true in M. The interpretation M is then a model for ϕ.

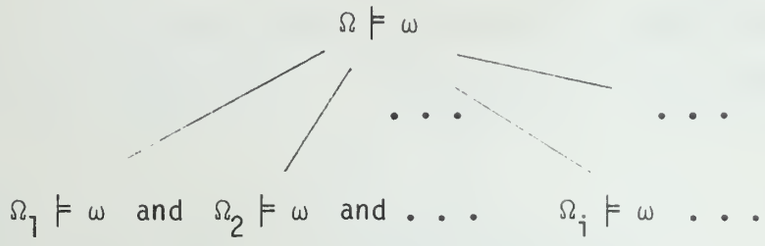
Def 4 A wff Ω logically implies a wff ω, notationally $\Omega \models \omega$, if ω is satisfied by every model of Ω.

Def 5 A wff ϕ_1 is equivalent to ϕ_2, $\phi_1 \models\!\models \phi_2$, if $\phi_1 \models \phi_2$ and $\phi_2 \models \phi_1$.

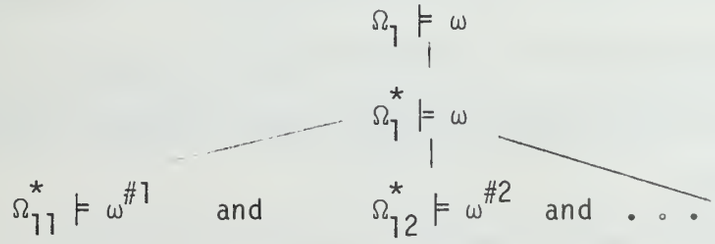
3.1.2 Outline of Theorem Prover

The general strategy of the proof procedure is as follows: The wffs Ω, and ω are in disjunctive normal form. If $\Omega \equiv \Omega_1$ or Ω_2 or . . .

then the proof of $\Omega \models \omega$ is a collection of proofs of $\Omega_i \models \omega$.



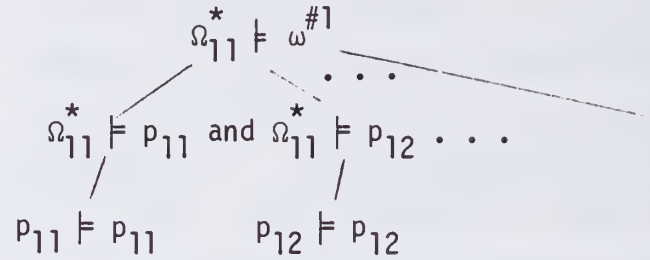
Given a disjunct Ω_1, and the conclusion ω to be made from it, we construct Ω_1^* from Ω_1 using certain "inference rules." (For our purposes, an inference rule is a procedure which transforms a given wff ϕ, according to certain criteria, into a wff ϕ' which is in a more convenient form than ϕ.) The wff Ω_1^* represents all that can be "deduced" out of the facts given by Ω_1, and is equivalent to Ω_1. However, Ω_1^* is not necessarily a single disjunct. Thus for each disjunct Ω_{1i}^* of Ω_1^*, we "normalize" Ω_{1i}^* and ω so that both wffs use not only the same set of ptrs but also use the same set of array segments. This may require "partitioning" the array segments they originally referred to into smaller segments. The wff ω is rewritten as $\omega^{\#1}$ using these smaller segments. Thus,



As we shall see later $\omega^{\#i}$ is equivalent to ω in the context of Ω_{1i}^*. The Ω_{1i}^*'s further have the property that if Ω_{1i}^* is satisfiable then

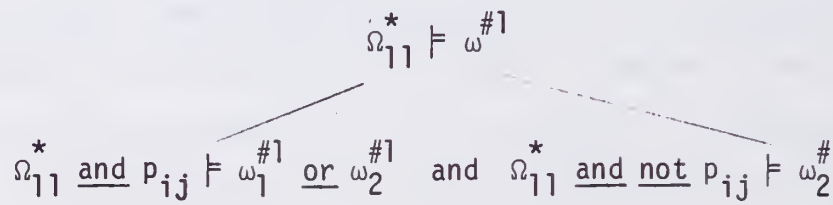
$$\Omega^*_{1i} \models p \quad \text{iff } P \models p \tag{3.1}$$

where P is a particular predicate of Ω^*_{1i} that "corresponds" to the predicate p of $\omega^{\#i}$. Thus, <u>if</u> $\omega^{\#1}$ is a disjunct,



where $\omega^{\#1} \equiv p_{11}$ <u>and</u> p_{12} <u>and</u> . . .

If $\omega^{\#1}$ is not a single disjunct, let it be $\omega^{\#1}_1$ <u>or</u> $\omega^{\#1}_2$ where $\omega^{\#1}_1$ is a single disjunct. Clearly, if $\Omega^*_{11} \models \omega^{\#1}_1$ then $\Omega^*_{11} \models \omega^{\#1}$. Otherwise, let p_{ij} be a predicate of $\omega^{\#1}_1$ which is not implied by Ω^*_{11}: $\Omega^*_{11} \not\models p_{ij}$. We then consider two cases of the premise:



We now take the transitive closure of the new premises Ω^*_{11} <u>and</u> p_{ij} to insure that property (3.1) mentioned above holds, and repeat the whole process for each of the new premises.

An example of Ω, ω and Ω^* and $\omega^{\#}$ is given in Figure 3.1.

3.1.3 <u>Inference Rules</u>

We now describe a procedure for generating the aforementioned Ω^*_{1i} and $\omega^{\#i}$ from Ω and ω. We will assume that Ω and ω are disjuncts. The more

$\Omega \equiv 2 < i + 1 < n$ and sorted$(1,i)$ and sorted$(i+1,n)$
$\omega \equiv$ array$(1,i) \leq$ array$(i+1,n)$ or $x_i > x_{i+1}$

array$(1,n)$ is partitioned into
array$(1,i-1)$; array(i,i);
array$(i+1,i+1)$; array$(i+2,n)$

$\Omega^* \equiv 2 \leq i$ and $i + 2 \leq n$ and $4 \leq n$ and
sorted$(1,i-1) \leq x_i$ and $x_{i+1} \leq$ sorted$(i+2,n)$

$\omega^{\#1} \equiv \omega_1^{\#1}$ or $\omega_2^{\#1}$ where

$\omega_1^{\#1}$ array$(1,i-1) \leq$ array$(i+2,n)$ and $x_i \leq$ array$(i+2,n)$ and
array$(1,i-1) \leq x_{i+1}$ and $x_i \leq x_{i+1}$

$\omega_1^{\#2} \equiv x_i > x_{i+1}$

The predicate $x_i \leq x_{i+1}$ of $\omega_1^{\#1}$ is not implied by Ω^*.
The two new premises are:

Ω^* and $x_i \leq x_{i+1}$ Ω^* and $x_i > x_{i+1}$

The transitive closure of Ω^* and $x_i \leq x_{i+1}$ is:

$2 \leq i$ and $i + 2 \leq n$ and $4 \leq n$ and
sorted$(1,i-1) \leq x_i$ and $x_{i+1} \leq$ sorted$(i+2,n)$ and
sorted$(1,i-1) \leq x_{i+1}$ and $x_i \leq$ sorted$(i+2,n)$ and
sorted$(1,i-1) \leq$ sorted$(i+2,n)$ and $x_i \leq x_{i+1}$

The predicate sorted$(1,i-1) \leq$ sorted$(i+2,n)$ of above "corresponds" to the predicate array$(1,i-1) \leq$ array$(i+2,n)$ of $\omega_1^{\#1}$.

Figure 3.1 An Example of Ω, ω, Ω^* and $\omega^{\#}$

general case will be dealt with later (see Algorithms 1, 2, 3 and 4 of Section 3.2). The procedure consists of several subprocedures (inference rules) each performing a distinct transformation on a subset of predicates of Ω and ω. We will find the following descriptions of the effect of applying the inference rules useful. We are interested only in "sound" inference rules:

Def 6 An inference rule r is sound if it yields ϕ' when applied on ϕ, notationally $\phi \vdash_r \phi'$, such that $\phi \models \phi'$.

Def 7 An inference rule r is information preserving iff ($\phi \vdash_r \phi'$ implies $\phi \models\!\models \phi'$).

Intuitively, no information carried by ϕ has been lost in the process of applying an information preserving rule.

Not all inference rules are information preserving. For example, our rule of local implication (Section 3.1.3) lets us conclude that $(u + k_2 \leq v)$ from $(u + k_1 \leq v)$ whenever $k_2 \leq k_1$. Clearly, this rule is not information preserving.

Def 8 An inference rule r is an enriching rule if it yields ϕ' when applied on ϕ such that

1. r is information preserving, and
2. For every $aR'b$ of ϕ', consider the predicates aRb of ϕ on the same variables a and b. Then $aR'b \models aRb$ but not necessarily $aRb \models aR'b$.

Note that an enriching rule does not actually create new information, but rather makes whatever information was present more readily usable. All our inference rules, except the abovementioned rule of local implication, are enriching rules.

It will be convenient to describe the rules on a directed graph representation of the wffs. The representation of a wff is the collection of the representations of its disjuncts. Each disjunct ϕ is represented as a pair of graphs--a ptr graph representing the conjunction of ptr predicates in the wff ϕ, and a key graph representing the array predicates. There is a coupling between these two graphs, namely, the boundaries of the array segments of the key graph are defined by the pointers.

The construction of the ptr graph π of a disjunct ϕ is described below. A partitioned array (key) graph will be constructed later.

3.1.3.1 Graph Construction

The ptr graph π will have a vertex for each ptr variable referred to in ϕ. For each ptr predicate $(u + k_1 \leq v)$ in ϕ, we put a directed edge from u to v and label it "k_1." Note that k_1 is a signed integer, and the relation is always $\leq$. The graph π may have more than one edge from a vertex u to a vertex v. An example of a disjunct ϕ and its pointer graph appears in Figure 3.2. In constructing the ptr graph the following equality axiom is embedded: for any ptr u, $u + 0 \leq u$.

3.1.3.2 Subsumption for Ptrs

The graph, as constructed in the preceding section, may have

$\phi \equiv 1 < j \leq i \leq n$ and array$(1,j-1) \leq x_j$ and

array$(1,i) \leq$ sorted$(i+1,n)$ and $j < i$

(ϕ is the premise of the verification condition V1 of Figure 2.3).

The ptr graph π of ϕ is

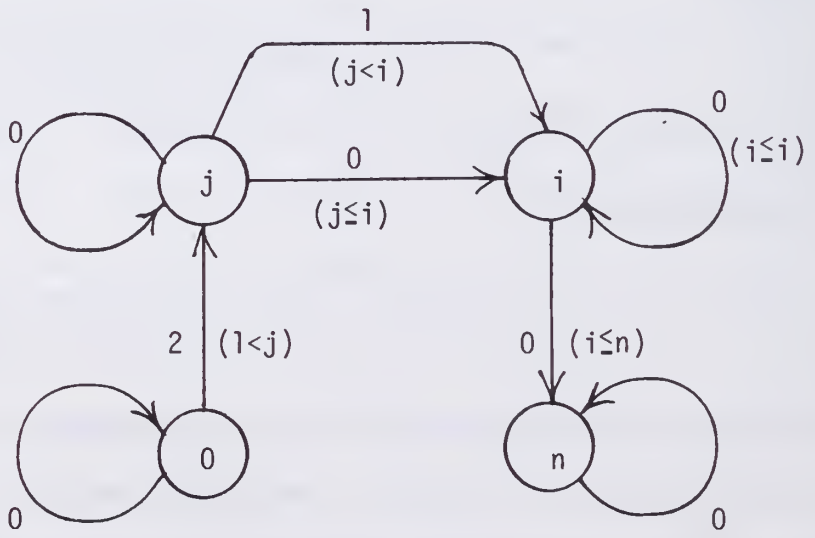


Figure 3.2 An Example of a Disjunct and Its ptr Graph

more than one directed edge from a vertex u to a vertex v. The rule of subsumption replaces all edges from vertex u to v by a single edge. The label on this single edge is a label on one of the previous edges (see Figure 3.3).

Let $\{k_1, k_2, \ldots, k_i\}$ be the set of labels (integer constants), on edges from u to v. Then we delete all these i edges, and replace them by a single edge with the label $k_{max} = \max \{k_1, k_2, \ldots, k_i\}$.

Remark 1: The rule of subsumption for ptrs is an enriching rule.

3.1.3.3 Transitivity Rule for Ptrs

For any pair of edges $(u + k_1 \leq v)$ and $(v + k_2 \leq w)$ of the ptr graph, add a new edge $(u + k_1 + k_2 \leq w)$.

By applying this rule to a ptr graph as long as it yields new edges, we construct the transitive closure of this ptr graph.

Remark 2: The transitivity rule is an enriching rule.

3.1.3.4 Partitioning of the Array

We now come to a simple, but powerful, idea of the theorem prover: express both premise and conclusion in terms of a common set of array segments. To do this, we partition the array (1,n) into nonoverlapping segments so that an array segment referred to in either of the wffs Ω or ω is the union of a few contiguous segments in the partition. Using these partitioned segments, equivalent wffs Ω^* and $\omega^\#$ are obtained.

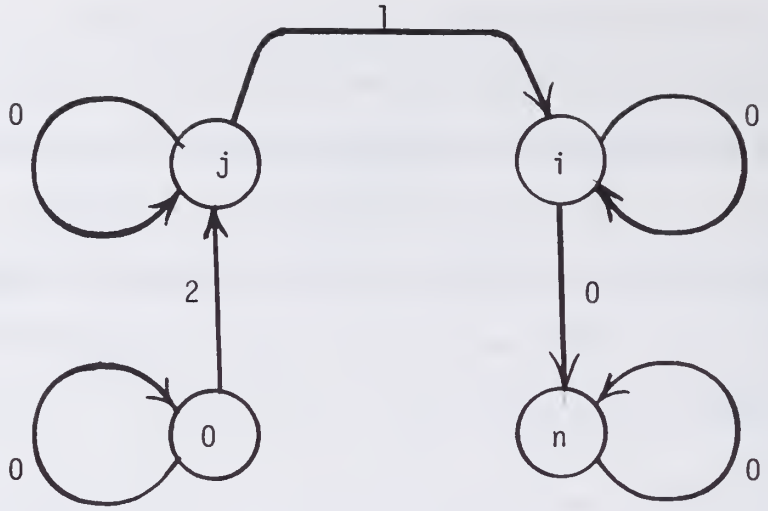


Figure 3.3 Ptr Graph of Figure 3.2 after Subsumption

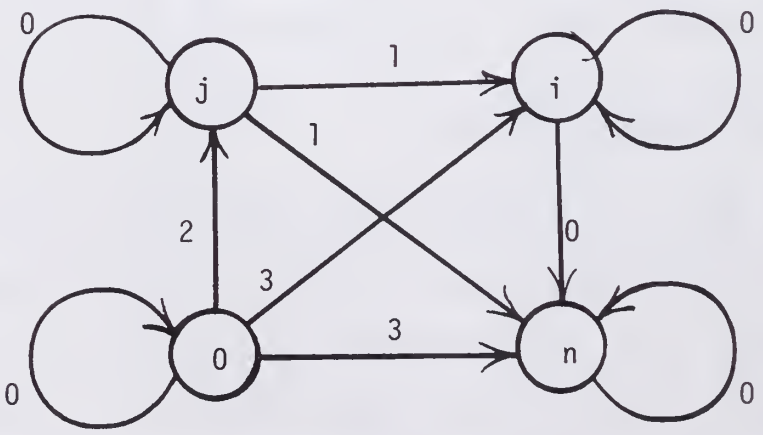


Figure 3.4 Transitive Closure π^* of the Ptr Graph π of Figure 3.3

Consider an array setment array (s,t). It partitions the entire array into three segments: $x_{-\infty} \cdots x_{s-1}$; $x_s \cdots x_t$; $x_t \cdots x_{+\infty}$, some of which may be empty. If we overlay one partition on another, we obtain their product. The array partition needed to express Ω and ω in terms of a common set of segments is obtained as the product of all individual partitions defined by the array segments occurring in either Ω or ω. Each linear ordering of the boundaries used in Ω or ω defines a partition of the array.

The partitioning procedure collects the relevant boundaries, and produces all linear orderings of these boundaries in the context of the partial ordering specified by the ptr graph π of Ω.

The set B of boundaries is constructed as follows:

Initially, $B \leftarrow \{-\infty, +\infty\}$

for each array segment array (s,t) referred to

either in Ω or in ω do

$B \leftarrow B \cup \{s-1, s, t, t+1\}$

endfor

Consider a maximal chain of the following kind, in the context of the given ptr graph π of Ω:

$$C : -\infty = b_0\ b_1\ b_2 \circ \cdots b_{2q}\ b_{2q+1} = +\infty$$

where for $1 \leq i \leq q$

b_{2i} and b_{2i+1} are boundaries in B, and

$b_{2i} = 1 + b_{2i-1}$, and

$b_{2i} \leq b_{2i+1}$

as implied by the ptr graph π of Ω

If every boundary b of B either appears in C, or is equal to a boundary appearing in C, then a composition (product) of the partitions is readily obtained:

$$b_0 \text{ to } b_1;\ b_2 \text{ to } b_3;\ \ldots;\ b_{2q} \text{ to } b_{2q+1}.$$

However, if for some boundary pair b and 1 + b of B at least one of them is not in C, then we resort to case analysis. We find the largest j such that

$$b_j \leq b$$

Clearly, it is not known if $b_{j+1} \leq b$ or $b_{j+1} > b$. For otherwise, we either have a larger j, or a longer chain. The wff Ω is equivalent to the disjunction of Ω_1, Ω_2 where

$$\Omega_1 \equiv \Omega \ \underline{\text{and}}\ (b < b_{j+1})$$

$$\Omega_2 \equiv \Omega \ \underline{\text{and}}\ (b \geq b_{j+1})$$

Proving $\Omega \models \omega$ is equivalent to proving

$$\Omega_1 \models \omega$$

and

$$\Omega_2 \models \omega,$$

and in each of Ω_1 and Ω_2 we can produce longer chains of boundaries than in Ω. We apply the same procedure of obtaining maximal chains of boundaries on each of $\Omega_1 \models \omega$ and $\Omega_2 \models \omega$. Clearly, this process will

terminate. Let $\Omega_1, \Omega_2, \Omega_3, \ldots, \Omega_C$ be the decompositions thus obtained; in each of Ω_i, the boundaries can all be put into one chain (see Figure 3.5). The following lemma immediately follows:

Lemma 1 Let $\Omega' \equiv \Omega_1$ or Ω_2 or . . . of Ω_C where Ω_i's are the result of decompositions of Ω made while ordering the boundaries. Then $\Omega \models \neg \Omega'$.

Note that the disjuncts $\Omega_1, \Omega_2, \ldots, \Omega_C$ differ only in the ptr predicates; they all have the same set of array predicates. The ptr graph of Ω defines a partial ordering on the set of boundaries B. From this partial ordering, say L, we are obtaining all linear orders $L_1, L_2, \ldots, L_C$. Hence

$$L \models \neg L_1 \text{ or } L_2 \text{ or } \ldots \text{ or } L_C.$$

If $\Omega \equiv S_\pi$ and S_α where S_π and S_α are respectively the set of ptr, and array predicates of the disjunct Ω, then

$$\Omega_i \equiv L_i \text{ and } S_\alpha.$$

Lemma 2 For $i \neq j$, (Ω and Ω_i and Ω_j) is unsatisfiable.

Construction of Key Graph

We construct the key graph α of a disjunct ϕ in the context of a given partition of the array defined by a linear ordering L of boundaries. For each segment array (s,t), there are two vertices: minx (s,t) representing the minimum key, and maxx (s,t) representing the maximum key of

Let $\omega \equiv 1 \leq i - 1 < n$ and array$(1,i-1) \leq$ sorted(i,n)

and $\Omega \equiv 1 < j \leq i \leq n$ and S_α and $j \geq \alpha$ where

$S_\alpha \equiv$ array$(1,j-1) \leq$ array(j,j) and array$(1,i) \leq$ sorted$(i+1,n)$

(The verification condition V2 in Figure 2.3 is $\Omega \models \omega$.)

The set of boundaries B = $\{-\infty,+\infty,0,1,i-1,i,n,n+1,j-1,j,j+1,i+1\}$

Maximal chain C induced by the ptr graph $\pi = 1 < j = i \leq n$ of Ω is:

$-\infty$ 0 1 i-1 i n n+1 $+\infty$

There are 2 linear orderings of boundaries in B

$\Omega \models \neg \Omega_1$ or Ω_2, where

$\Omega_1 \equiv 1 < j = i < n$ and S_α

C_1 : $-\infty$ 0 1 i-1 i i+1 n n+1 $+\infty$

partition : array$(-\infty,0)$; array$(1,i-1)$; array(i,i)

array$(i+1,n)$; array$(n+1,+\infty)$

$\Omega_2 \equiv 1 < j = i = n$ and S_α

C_2 : $-\infty$ 0 1 i-1 i i+1 $+\infty$

partition : array$(-\infty,0)$; array$(1,i-1)$

array(i,i); array$(i+1,+\infty)$

Figure 3.5 An Example of Ordering Boundaries and Partitioning the Array

the segment. The edges in the key graph are labeled using the subarray relationships induced by the array predicates S_α of the disjunct ϕ. The following axioms are embedded in the construction of this key graph α from the set S_α of unpartitioned array predicates.

- If array (s,t) is nonempty, then minx (s,t) $\leq$ maxx (s,t)
- A subsegment of a sorted array segment is sorted
- If array (s,t) and array (u,v) are subsegments of a sorted segment and if t $\leq$ u then array (s,t) $\leq$ array (u,v)
- If s = t then array (s,t) $\leq$ array (s,t)
- If two array segments are related by R, then their respective subsegments are also R-related.

An array segment array (s1,t1) is a subsegment of array (s,t) if s $\leq$ s1 $\leq$ t1 $\leq$ t. The sorted (s,t) predicate is represented as a pseudo-binary relation: array (s,t) sorted array (s,t). These axioms are used to put labeled edges between vertices of the key graph : for each array predicate (array (s,t) R array (u,v)), we put an edge from maxx (s,t) to minx (u,v) and label it with R (see Figure 3.6). We do not construct a key graph in the absence of a partition. Thus, when considering a key graph a "context" is always present.

Remark 3: α and L $\models \exists S_\alpha$ and L, where L is the linear ordering of boundaries which defined the present partition.

Remark 4: The negation of the predicate sorted (s,t) is not representable. In Section 3.2.2, we prove (Theorem 4) that representing sorted (s,t) will not be necessary.

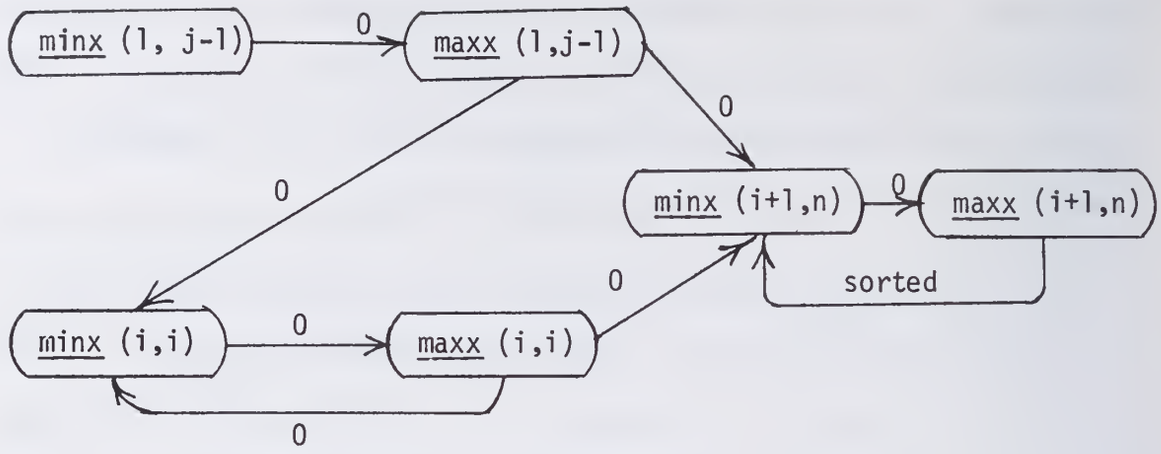


Figure 3.6 Key graph for S_α in the context of the Partition C_1 of Figure 3.5

3.1.3.5 Rule of Subsumption for Array Predicates

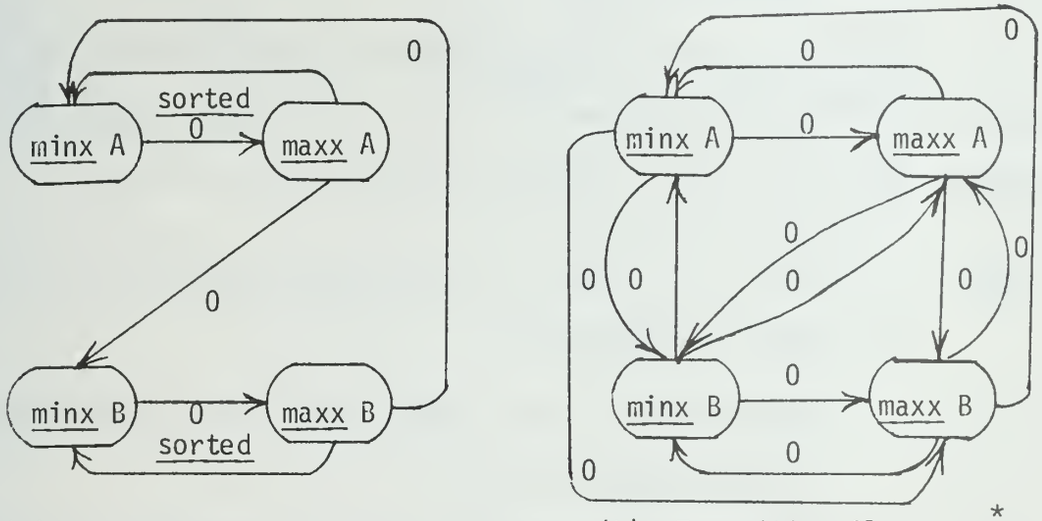
The rule of subsumption replaces all edges from a vertex v_1 to v_2 by a single edge. The label on this single edge is a label on one of the previous edges.

> Let $\{k_1, k_2, \ldots, k_i\}$ be the set of labels (integer constants and sorted) on edges from v_1 to v_2. Then we delete all these i edges and replace them by a single edge with the label $k_{max} = \max \{k_1, k_2, \ldots, k_i\}$. For the purpose of this rule we define the label sorted to be less than any other label.

The rule is identical to the rule of subsumption for pointer predicates. Note that maxx (s,t) $\leq$ minx (s,t) implies sorted (s,t) since all the elements of array (s,t) are then equal, while the converse is not true. For

this reason, we defined the label sorted to be the smallest label (see Figure 3.7).

Remark 5: The rule of subsumption for array predicates is an enriching rule.



(a) A Key Graph α

(b) Transitive Closure α^* of Key Graph α (self-loops not shown)

Figure 3.7 A Key Graph and Its Transitive Closure

3.1.3.6 Transitivity Rule for Array Segments

The transitive closure of key graph α is obtained in a way similar to that of ptr graph π.

If $(v_1 + k_1 \leq v_2)$ and $(v_2 + k_2 \leq v_3)$ then
$(v_1 + k_1 + k_2 \leq v_3)$

Recall that the labels k can be either integer constants, or sorted. For the purpose of this rule, we define the label sorted to satisfy:

1. sorted < any other label
2. sorted + any label = sorted = any label + sorted

Remark 6: The transitivity rule for array segments is an enriching rule.

3.1.3.7 Rule of Local Implication

This rule lets us conclude a global property like $\phi \models p$, where p is a ptr or key predicate, and ϕ is in a certain form, from a local property that $p' \models p$, the p' being a particular predicate of ϕ.

Def 11 A disjunct $\phi^* \equiv \pi^*$ and α^* is an enriched disjunct with respect to a set B of boundaries if

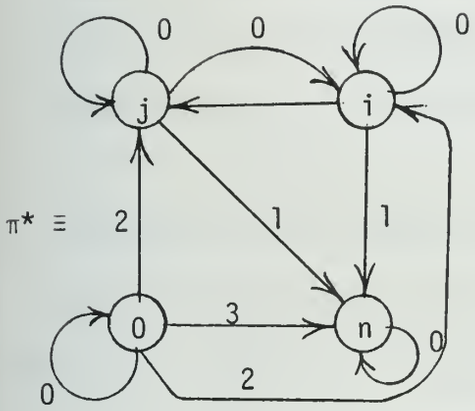
1. The ptr graph π^* of ϕ^* defines exactly one linear ordering L of the boundaries of B
2. The array predicates of α^* have been expressed using the segments defined by the partition induced by the linear ordering L of the boundaries
3. Both π^*, and α^* are transitively closed

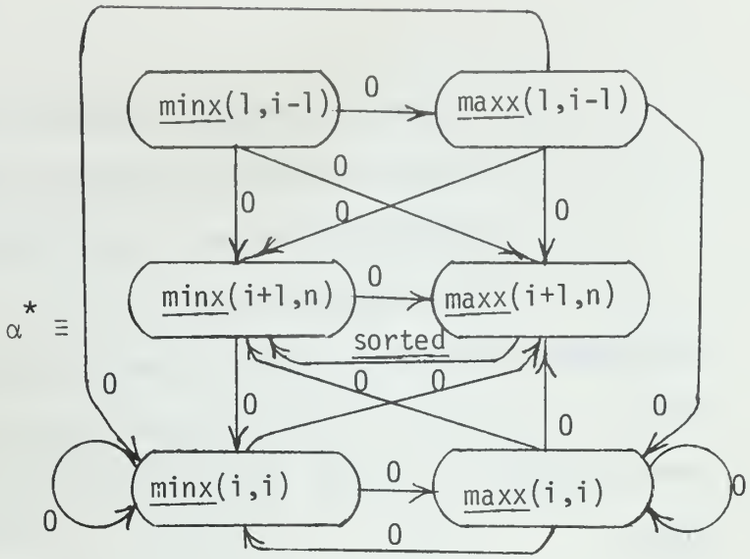
Let $\phi_1^* \equiv \pi_1^*$ and α_1^* be an enriched disjunct. Let $\phi_2 \equiv \pi_2$ and α_2 be a disjunct such that the vertex-set of π_2 is the same as that of π_1^* and the vertex-set of α_2 be the same as that of α_1^*.

Def 12 For each predicate (u R_2 v) of ϕ_2, the corresponding predicate in ϕ_1^* is defined as follows:

1. If there is an edge from u to v labeled with, say, R_1 in α_1^*, the corresponding predicate is (u R_1 v).

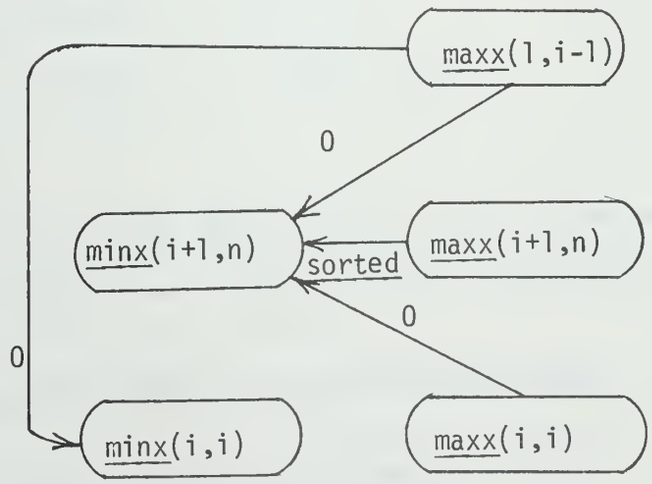
$\Omega_1^* \equiv \pi^*$ and α^*, where





α^* is the transitive closure of Figure 3.6

The conclusion $\omega^{\#1}$ in the context of the partition defined by Ω_1^* is



$\omega^{\#1}$ immediately follows from Ω_1^* by the rule of local implication

Figure 3.8 An Enriched Disjunct Ω_1^* of Ω of Figure 3.5

2. If there is no edge from u to v in α_1^* then the corresponding predicate is (u null v), an empty predicate, which is defined to be true in all interpretations. (Intuitively, take null as "$-\infty \leq$."

Consider a disjunct ω_1 of the conclusion ω to be made from an enriched disjunct Ω^*. Let $\omega_1^\#$ be the rewritten version of ω_1 using the partitioned array segments. Since $\omega_1^{\#1}$ is equivalent to ω_1, in this context, it follows that

$$\Omega^* \models \omega_1 \text{ iff } \Omega^* \models \omega_1^\#$$

$$\text{iff } \Omega^* \models p \text{ for every predicate p of } \omega_1^\#$$

Because Ω^* is an enriched disjunct, we can make the following stronger statement

$$\Omega^* \models \omega_1^\# \quad \text{iff} \quad \text{for any predicate p of } \omega_1^\#, \text{ the corresponding predicate P of } \Omega^* \text{ is such that } P \models p \qquad (3.2)$$

We shall refer to (3.2) as the rule of local implication. A proof of the validity of this rule is given in Section 3.2.2.

3.2 Basic Theorem Prover is a Decision Procedure

An informal, but complete, description of the basic theorem prover is contained in Algorithms 1, 2, 3 and 4 in the following pages. This section proves that the theorem power is a decision procedure for $\Omega \stackrel{?}{\models} \omega$, where both Ω and ω are wffs. The theorem prover will be extended in Section 3.3 to prove $\Omega \models \omega$ where either or both of Ω and ω can be augmented wffs.

That the basic theorem prover terminates follows immediately by considering the "length" of the conclusion ω. In Algorithm 3, and 4, we delete either a whole disjunct, or a predicate of a disjunct from ω in every iteration. In the present section, we shall be occupied with the proof that the basic theorem prover gives correct answers, that is,

> when the basic theorem prover terminates, the boolean variable - istheorem - is <u>true</u> iff it is indeed the case that the premise Ω logically implies the conclusion ω. (3.3)

The core of the proof is that the rule of local implication is valid. The structure of the proof follows the structure of the algorithms closely. We show (1) that the satisfiability of a graph is decidable and that it is obtained as a by-product of transitive closure, and (2) that $\Omega^* \models p$ iff p follows from Ω^* by local implication.

3.2.1 Model Construction

Given an enriched disjunct $\phi^* \equiv \pi^*$ <u>and</u> α^*, we want to construct a model for it by assigning values to ptrs and keys.

Without loss of generality, we can assume that the vertex 0 is present in the ptr graph π^*, for, if it is not, introduce it by <u>and</u>ing π^* with $0 \leq 0$. This vertex is then assigned the value zero. A model for π^* is constructed first, and then a model for α^* is similarly constructed.

```
procedure basic theorem prover (Ω : wff ⊨ ω : wff)
    istheorem ← true
    if ω is empty then ω ← false endif
    for each disjunct Ω1 of Ω and while istheorem do
        provetheorem  ( Ω1 ⊨ ω)
    end for-while
endproc
```

Algorithm 1

```
procedure provetheorem  (Ω1 : disjunct ⊨ ω : wff)
    construct the ptr graph π of Ω1; apply subsumption.
    if π is satisfiable
    then {see Algorithm 5 of Section 3.3.1}
        collect the boundaries referred to in Ω1, and ω into
            a set B.
        for each linear ordering L, according to π, of
            boundaries of B
        and while istheorem do
            construct the key graph α of Ω, for this parti-
                tion defined by L
            apply subsumption on α
            istheorem ← provelemma (π and L and α ⊨ ω)
        end for-while
    endif
endproc
```

Algorithm 2

```
function provelemma (Ω11 : disjunct ⊨ reference ω' : wff)
returns value of local islemma : boolean
    π* ← transitive closure of ptr graph of Ω11
    α* ← transitive closure of key graph of Ω11
    islemma ← (π* and α* is unsatisfiable)
    if not (ω' is empty or islemma) then
        choose a disjunct ω1 of ω'; delete ω1 from ω'
        ω1# ← equivalent of ω1 expressed using the partition
            defined
        islemma ← does (π* and α*) imply (ω1# or ω')?
    endif
endfunction
```

Algorithm 3

function does (Ω_{11}^{*} : enriched disjunct) imply ($\omega_1^{\#}$: partitioned disjunct or reference ω' : wff)?

returns value of local implies : boolean

    repeat

        choose a predicate p of $\omega_1^{\#}$; delete p from $\omega_1^{\#}$

    until $\omega_1^{\#}$ is empty or $\Omega_{11}^{*} \not\models p$

    if $\Omega_{11}^{*} \models p$ {local implication}

    then implies ← true

    else implies ← provelemma (Ω_{11}^{*} and $p \models \omega_1^{\#}$ or ω') and
                   provelemma (Ω_{11}^{*} and not $p \models \omega'$)
                   {see Theorem 4 of Section 3.2.2}

    endif

endfunction

Algorithm 4

Model for π^*

The model for π^* is constructed iteratively. Let

assigned = subset of vertices of π^* to which values are already assigned, such that for u_1, $u_2 \in$ assigned $(u_1 + k_1 \leq u_2)$ of π^* is satisfied, i.e., valueof $(u_1) + k_1 \leq$ valueof (u_2).

This model is then extended by choosing an arbitrary vertex v of π^*, which is not in assigned. If no such vertex exists, then the construction of a model for π^* is finished, and π^* is satisfiable. Consider the following set:

$$S = \text{predicates of } \pi^* \text{ to be satisfied by } v$$
$$= \{(u_i + k_i \leq v) \mid u_i \in \text{assigned}\} \cup \{(v + k_j \leq u_j) \mid u_j \in \text{assigned}\}$$

The value to be assigned to v should be such that all the predicates in S are indeed satisfied, and hence

assigned $\leftarrow$ assigned $\cup \{v\}$,

provided the label on the self-loop at v is zero.

Let V_{max} = max of valI and V_{min} = min of valJ, where

$$\text{valI} = \{\text{valueof } (u_i) + k_i \mid (u_i + k_i \leq v) \in S\} \cup \{-\infty\}$$

$$\text{valJ} = \{\text{valueof } (u_j) - k_j \mid (v + k_j \leq u_j) \in S\} \cup \{+\infty\}$$

If ptr v is assigned a value V such that

$$V_{max} \leq V \leq V_{min} \tag{3.4}$$

then all the predicates in S are satisfied. However, if the self-loop at v, $(v + k \leq v)$, is such that the label $k > 0$, this predicate is not satisfiable in any interpretation, and any model construction cannot proceed further.

Now, assuming that the self-loop at v is labeled with zero, we show that a value V satisfying the inequality (3.4) must exist. Let V_{max} be valueof $(u_1) + k_1$ (without loss of generality on the subscripts i of u_i), and V_{min} be valueof $(u_2) - k_2$. Thus

$$(u_1 + k_1 \leq v) \in S \text{ and } (v + k_2 \leq u_2) \in S.$$

Since π^* is transitively closed, $(u_1 + k' \leq u_2) \in \pi^*$ for some k' greater than or equal to $k_1 + k_2$, and since u_1 and u_2 are in assigned, we have

$$(u_1 + k' \leq u_2) \in S$$

by our hypothesis on the set assigned,

$$\text{valueof } (u_1) + k' \leq \text{valueof } (u_2)$$

and, therefore

$$V_{max} \leq V_{min}.$$

The vertex v can, therefore, be assinged any finite value in the range $[V_{max}, V_{min}]$.

Model for α^*

The boundaries of array segments are defined by ptrs and hence a model for α^* can be constructed only after a model for the ptrs is given. If array (s,t) is a segment used in α^*, we have, in general,

$$\underline{minx}\ (s,t) \leq \underline{maxx}\ (s,t)$$

If s = t then this becomes an equality. We remark that the labels or self-loops of α^* cannot be negative. A positively labeled self-loop is clearly unsatisfiable. Thus, these labels can only be either sorted or zero. For each pair of vertices minx (s,t), maxx (s,t) of α^*, we will assign a single value. This assignment clearly satisfies all self-loops, and sorted predicates. Once this decision is made, the model construction for α^* is identical to that for π^*.

3.2.2 Unsatisfiability and Local Implication Theorem

Theorem 1 (Unsatisfiability Theorem)

The ptr graph π is unsatisfiable iff the transitive closure π^* of π has a self-loop whose edge label is positive.

Proof We know from Remarks 1 and 2 that

$$\pi \models \models \pi^*$$

Thus, π is unsatisfiable iff π^* is.

($\Leftarrow$) If π^* has a self-loop at v with a positive label k, clearly $(v + k \leq v)$ is unsatisfiable in any interpretation.

($\Rightarrow$) It is well known that the transitive closure G^* of a graph G will have a self-loop at a vertex v iff G has a directed cycle (not necessarily of length 1) passing through v. The rule of transivity is such that the label on the self-loop at v in π^* is not less than the run of labels of edges in any directed cycle of π passing through v. Thus, if π has no directed cycle with positive edge-label sum passing through v, then π^* does not have a self-loop at v with a positive label. For such a π^*, we can indeed construct a model (see Section 3.2.1), and hence π is satisfiable. ∎

Corollary to Theorem 1 (Unsatisfiability of Key Graphs)

The key graph α, in the context of an enriched ptr disjunct, is unsatisfiable iff the transitive closure α^* of α has a self-loop whose edge label is positive.

Theorem 2 (Local Implication Theorem)

Let π^* be a transitively closed ptr graph. Then $\pi^* \models (u + k_2 \leq v)$ iff the corresponding predicate $(u + k_1 \leq v)$ in π^* is such that $k_1 \geq k_2$.

Proof

($\Leftarrow$) Obvious.

($\Rightarrow$) We prove that if $k_1 < k_2$ then $\pi^* \not\models (u + k_2 \leq v)$. Assign u an

arbitrary value, and then assign v a value equal to valueof (u) + k_1. Set assigned←{u,v}. Now, we can complete the construction of the model as in Section 3.2.1. Clearly, $(u + k_2 \leq v)$ is false in this model. ∎

Corollary to Theorem 2

Theorem 2 holds for a transitively closed key graph α^*, and array predicate $(u + k_2 \leq v)$.

Theorem 3

Let Ω^* be an enriched disjunct, and $\omega^\#$ a disjunct partitioned with respect to the linear ordering of boundaries defined by Ω^*. Then

$\Omega^* \models \omega^\#$ iff either, for every predicate p of $\omega^\#$, Ω^* locally implies p, or Ω^* is unsatisfiable

Proof by Theorem 1 and repeated application of Theorem 2. ∎

When an enriched disjunct π^* and α^* does not imply a predicate p of $\omega_1^\#$, we consider two cases (refer to Algorithm 4, if-statement):

$$\pi^* \text{ and } \alpha^* \text{ and not } p \models \omega \tag{3.5}$$

$$\pi^* \text{ and } \alpha^* \text{ and } p \models (\omega_1^\# \text{ or } \omega) \tag{3.6}$$

If p is sorted (s,t), not p cannot be represented in our scheme (Section 3.1.3.4). Hence a proof/disproof of (3.5) cannot be obtained in this

deductive system. The following theorem avoids this problem by showing that if p is a sorted-predicate, then the proof of (3.5) is equivalent to the proof of (π^* and s < t) and α^* and minx (s,t) < maxx (s,t) $\models \omega$ when $\alpha^* \not\models$ a sorted-predicate, p. ∎

Theorem 4

Let ϕ^* be an enriched disjunct and array (s,t) be a segment in the partition defined by ϕ^*. Further assume that $\phi^* \not\models$ sorted (s,t). Then

$$\phi^* \text{ and not sorted } (s,t) \models \psi^{\#} \text{ iff } \phi = \psi^{\#}$$

where $\psi^{\#}$ is a partitioned wff in the context of ϕ^*, and $\phi' = \phi^*$ and s < t and minx (s,t) < maxx (s,t)

Proof

($\Leftarrow$) If $\phi^* \models \psi^{\#}$ then ϕ^* and not sorted (s,t) = $\psi^{\#}$ is obvious.

($\Rightarrow$) Suppose ϕ^* and not sorted (s,t) $\models \psi^{\#}$. If ϕ^* is unsatisfiable, or if array (s,t) is empty, the theorem is trivially true. So let ϕ^* and not sorted (s,t) be satisfiable. Consider any model M for ϕ^*, $\models_M \phi^*$. If sorted (s,t) is false in this model, then $\models_M \psi^{\#}$. Given a model, any permutation of elements of sorted (s,t) conserves the minx (s,t) and maxx (s,t). Thus, if we permute the elements of array (s,t) in model M, all predicates of $\psi^{\#}$, with the possible exception of (array (s,t) R array (s,t))-type predicates, must still be true. Since $\psi^{\#}$ is a wff (in our

system) there are only three possibilities for (array (s,t) R array (s,t)):

1. array (s,t) < array (s,t)
2. array (s,t) ≤ array (s,t)
3. array (s,t) sorted array (s,t)

The first one is unsatisfiable in every interpretation. The second one will still be true after permutation. The third one was false in M, and if the permutation is an appropriate one it may be true in the resulting model M'. But, if $\psi^{\#}$ had this sorted (s,t) predicate, ϕ^{*} and not sorted (s,t), being a satisfiable disjunct, cannot imply $\psi^{\#}$. Thus, if $\models_{M} \phi^{*}$ and not sorted (s,t) then $\psi^{\#}$ will be true in a model M' also where M' is the result of permuting elements of array (s,t). That is, $\psi^{\#}$ will be true in any model for ϕ^{*}. ∎

3.2.3 Basic Theorem Prover is Correct

The structure of the proof of $\Omega \models \omega$, as constructed by this theorem prover, is shown in Figure 3.9. Theorem 3 yields the proof at the lowest level in Figure 3.9; the remaining proofs are proven by appropriate recursive/iterative calls (indicated by dotted lines; see Algorithm 1, 2, 3, and 4).

We omit further details of the correctness proof of the basic theorem prover.

Let $\Omega \equiv \Omega_1$ <u>or</u> Ω_2, where Ω_1 is a disjunct, and Ω_2 is a wff, possibly <u>false</u>

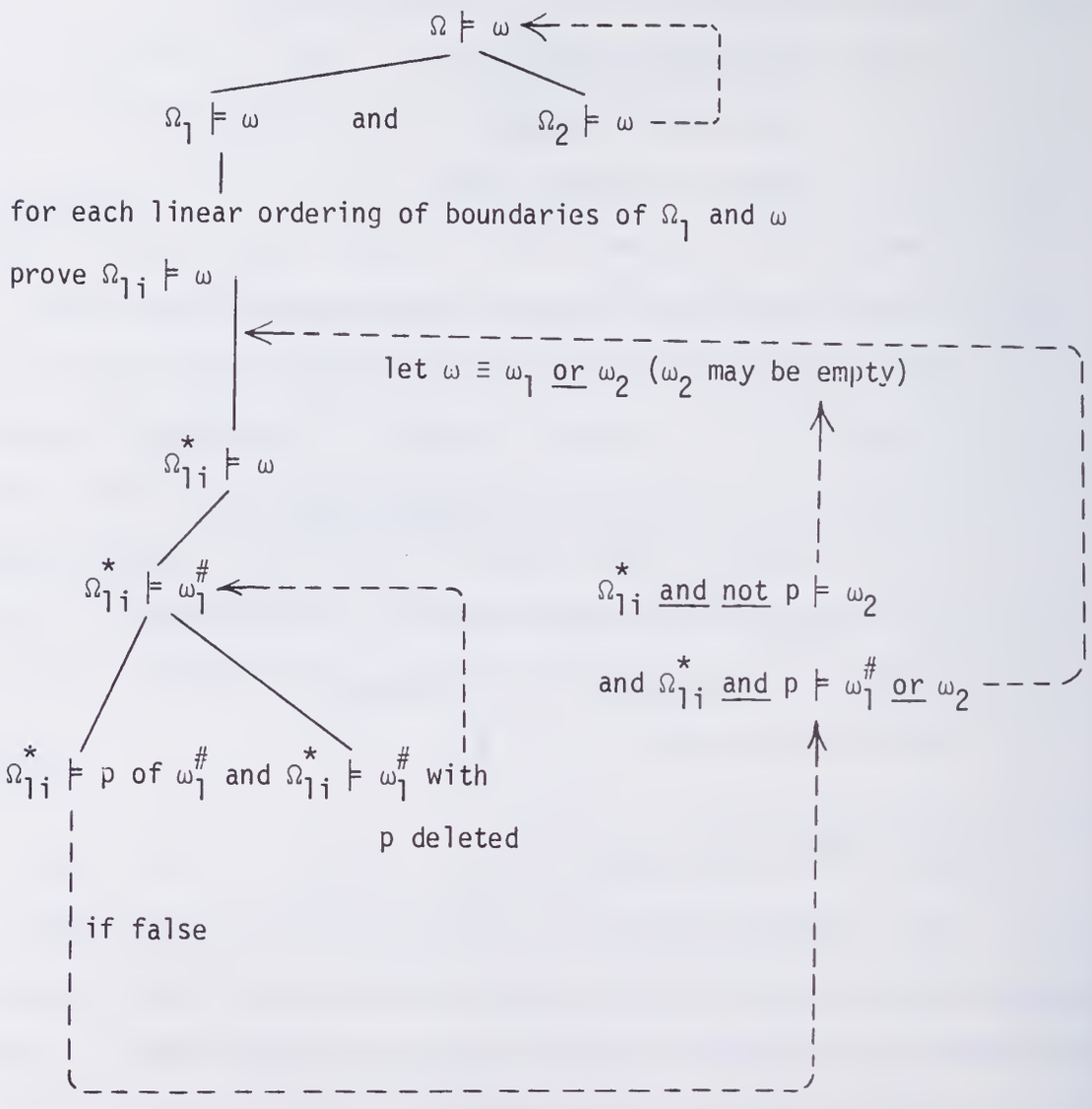


Figure 3.9 Structure of the Proof of $\Omega \models \omega$

3.3 Evaluation of Backward Functions

Recall that the conclusion ω of the lemmas $\Omega \models \omega$ to be proven was an augmented wff, possibly involving the functions subst, exchb and nsrtb in because of backward substitution (see Chapter 2). Similarly, Ω was an augmented wff possibly involving the functions subst and unmodifiedpartsof. To be able to use the basic theorem prover presented in Section 3.1, we transform these augmented wffs by evaluating the functions to produce simple wffs not involving any of these functions.

Strictly speaking, the evaluation of functions like exchb, nsrtb, etc., cannot be considered part of theorem proving. However, we include it here because it plays an important role in our theorem proving, and because this evaluation is done in the midst of the theorem-proving effort.

Given the lemma $\Omega \models \omega$ to be proven, the subst functions, if any, of Ω and ω are evaluated first. Let us call the resulting augmented wffs Ω and ω. The premise Ω can be considered to be Ω_1 or Ω_2 where Ω_1 is a disjunct, and Ω_2 is an augmented wff, possibly the wff false. We then prove that $\Omega_1 \models \omega$, and that $\Omega_2 \models \omega$. In Section 3.3.2, we describe the proof of $\Omega_1 \models \omega$. (This procedure is used repeatedly for each disjunct of Ω.) The boundaries of Ω_1 and ω are collected into a set B. Each linear ordering of boundaries defines a partition of the array. The boundaries collected are such that this partition has single element array segments array(s,s) and array (t,t) for each exchange x_s with x_t statement (similarly for insert statements). It is then a simple matter to evaluate the exchb and nsrtb functions. The resulting $\dot{\omega}$ is a simple wff.

We now describe these two passes of evaluation in greater detail.

3.3.1 First Pass of Evaluation

The evaluation of the subst functions is the simplest, and constitutes the first pass of our evaluation. Clearly, before

subst t for u in ψ

can be evaluated, all subst functions of the augmented wff ψ must be evaluated. Assuming that ψ is free of subst functions, the ptr expression t is substituted for every occurrence of the ptr variable u in the augmented wff ψ, which may have only exchb and nsrtb functions.

Remark 7: Let S be $u \leftarrow t$ statement. Then, the entry assertion $\phi_E \equiv$ subst t for u in ψ_S generated for an exit assertion ψ_S is such that

if $\models_I \phi_E$ then $\models_{I'} \psi_S$, and

if $\not\models_I \phi_E$ then $\not\models_{I'} \psi_S$

where I' is the result of execution of S on an interpretation I.

The boundaries referred to in the conclusion ω and current disjunct of the premise Ω_1 are collected by Algorithm 5. As can be easily seen, the boundaries included in B are such that the partition produced is guaranteed to contain appropriate segments needed in the evaluation of exchb, nsrtb and unmodifiedpartsof in the second pass.

$B \leftarrow \{-\infty, +\infty\}$

for each array segment array (s,t) referred to either in
    Ω or in ω do
    $B \leftarrow B \cup \{s-1,s,t,t+1\}$
endfor

for each exchb x_s with x_t in ψ occurring in ω do
    $B \leftarrow B \cup \{s-1,s,s+1,t-1,t,t+1\}$
endfor

for each nsrtb x_s below x_t in ψ occurring in ω do
    $B \leftarrow B \cup \{s-1,s,s+1,t-2,t-1,t,t+1\}$
endfor

for each unmodifiedpartsof α wrt (s,t) occuring in Ω_1 do
    $B \leftarrow B \cup \{s-1,s,t,t+1\}$
endfor

Algorithm 5: Collecting Boundaries

3.3.2 Second Pass of the Evaluation

The second pass is made for each linear ordering L of the boundaries collected as above. None of the functions exchb, nsrtb and unmodifiedpartsof changes the ptr expressions. While the first pass has an effect only on ptr expressions, the second pass has its effect only on the array segments, which depend on the context L. Again, the evaluation of exchb and nsrtb is from inside out.

exchb

Assuming that ψ in a wff, that is, that ψ is free of exchb and nsrtb functions,

$$\underline{\text{exchb}}\ x_s\ \underline{\text{with}}\ x_t\ \underline{\text{in}}\ \ \psi$$

is evaluated in the context of the partition defined by the current linear ordering of boundaries. The wff ψ is expressed as $\psi^{\#}$ using the partitioned array segments. Note that the partition produced will have single element array segments A ≡ array (s,s), and B ≡ array (t,t) (see, second for-loop of Algorithm 5). The exchb is evaluated by substituting B for A, and vice-versa, in every array predicate of $\psi^{\#}$.

nsrtb

Again assuming that ψ is free of exchb and nsrtb functions,

$$\underline{\text{nsrtb}}\ x_s\ \underline{\text{below}}\ x_t\ \underline{\text{in}}\ \psi$$

is evaluated in the context of the present partition. The wff ψ is expressed as $\psi^{\#}$ using the partitioned array segments. Note that since s-1, s, s+1, t-2, t-1, t and t+1 are included in the set of boundaries, the partition produced will have single-element segments array (s,s), array (t-1,t-1), and array (t,t). The boundaries are already ordered, and we consider the two cases $s < t$, or $s \geq t$.

Suppose $s < t$. Then the following transformations are made on the array segments A ≡ array (u,v) of the predicates of $\psi^{\#}$:

1. If A is a subsegment of array (s,t-2) then A is redefined as array (u+1,v+1).
2. If A is the same segment as array (t-1,t-1) then A has the new definition: array (s,s).
3. The definition of A is unchanged otherwise.

Now suppose $s \geq t$. Then the following transformations are made on the array segment A:

1. If A is a subsegment of array (t+1,s) then A is redefined as array (x-1,y-1).
2. If A is the same segment as array (t,t) then A is redefined as array (s,s).
3. Otherwise, the definition of A is unchanged.

Remark 8: Let S be either an exchange x_s with x_t or an insert x_s below x_t statement, and let ϕ_E be the corresponding exchb x_s with x_t in ψ_S or nsrtb x_s below x_t in ψ_S statement where ψ_S is the exit assertion of S. Then ψ_S is true in M', which is the result of the execution of S on M, iff ϕ_E is true in M, where M is a model for the context. More formally, if L is the present linear ordering of boundaries and $\models_M L$ then

if $\models_M \phi_E$ then $\models_{M'} \psi_S$, and

if $\not\models_M \phi_E$ then $\not\models_{M'} \psi_S$

Lemma 3 Let S be a straightline program segment, that is, S is a sequence of ptr assignment, exchange or insert statements, with ψ_S as its exit assertion, and let ϕ_E be the entry assertion of S obtained as above

in the context of L. Then

$$\text{if } \models_M \phi_E \text{ then } \models_{M'} \psi_S, \text{ and}$$

$$\text{if } \not\models_M \phi_E \text{ then } \not\models_{M'} \psi_S$$

where M is a model for the linear ordering L of boundaries and M' is the result of execution of S on M.

Proof by repeated applications of Remarks 7 and 8.∎

unmodifiedpartsof

The description of the evaluation of

unmodifiedpartsof α wrt (s,t)

is somewhat complicated because of the details needed. Intuitively, since the procedure called can permute the elements of array (s,t), all predicates of $\alpha^{\#}$ which depend on the strict subsegments of array (s,t) are deleted from $\alpha^{\#}$. In addition, the predicate sorted (s,t), if present in $\alpha^{\#}$, is deleted from $\alpha^{\#}$. The complication arises from the possibility that the current linear ordering of boundaries partitions the segment array (s,t) into smaller segments. In such a case, it will be necessary to temporarily "join together" contiguous segments to see if the entire segment array (s,t) is related to other segments of array $(-\infty,s-1)$ or array $(t+1,+\infty)$.

Let array (s,t) $\equiv S_1; S_2; \ldots; S_p$; that is, the S_is constitute the p subsegments of array (s,t) from the boundary s to t. Then the evaluation is done as described in Algorithm 6.

$\phi \leftarrow$ the wff *false*

for each disjunct α_1 of α *do*

$\pi_1 \leftarrow$ ptr graph of α_1

for each linear ordering L induced by π_1^* *do*

$\phi_1 \leftarrow L$

let $\alpha_1^{\#}$ be the disjunct α_1 expressed using partitioned segments

for each array predicate (ARB) of $\alpha_1^{\#}$ *do*

cases

neither A nor B is an Si: $\phi_1 \leftarrow \phi_1$ *and* (ARB) provided (ARB) is not *sorted* A

A is an Si, B is not : $\phi_1 \leftarrow \phi_1$ *and* (*array* (s,t) $R'B$) provided $\alpha_1^{\#}$ has predicates S_1R_1B, S_2R_2B, . . ., S_pR_pB such that for $1 \leq j \leq p$, $S_jR_jB \not\models S_jR'B$ and if $S_jR_jB \not\models S_jR''B$ then $S_jR'B \not\models S_jR''B$

B is an S_i, A is not : as above with A, B exchanged

A is an S_i, B is an S_j : ϕ_1 is unchanged

endcases

endfor

$\phi \leftarrow \phi_1$ *or* ϕ

endfor

endfor

Algorithm 6: Obtaining $\phi \equiv$ *unmodifiedpartsof* α *wrt* (s,t)

This completes the description of the evaluation of functions. In the next section, we present the theorem prover with the extensions required by the evaluation, and give arguments to establish the fact that the extended theorem prover is a decision procedure.

3.4 Extended Theorem Prover is a Decision Procedure

Let us briefly review the backward substitution method (Section 2.2.2) of generating the verification conditions. To prove $\{\Phi|P|\Psi\}$, the program P is decomposed into straightline program segments S and we then prove $\{\phi|S|\psi\}$, where ϕ and ψ are generated from Ψ, and the loop invariants given. Each $\{\phi|S|\psi\}$ is proven by proving the generated lemma $\phi \models \phi_B$, where ϕ_B is the entry assertion for S generated from ψ by backward substitution. We recall that the backward substitution of [King 1969] is such that

$$\begin{aligned} &\text{if } \models_I \phi_B \text{ then } \models_{I'} \psi, \text{ and} \\ &\text{if } \not\models_I \phi_B \text{ then } \not\models_{I'} \psi \end{aligned} \tag{3.7}$$

where I is any interpretation, and I' is the result of executing the program segment S on I. Note that $\{\phi_B|S|\psi\}$ is a milder statement tha (3.7). It follows immediately that, for any entry assertion ϕ,

$$\{\phi \mid S \mid \psi\} \text{ iff } \phi \models \phi_B \tag{3.8}$$

In general, ϕ_B has many disjuncts which are unsatisfiable in every model

of ϕ, making it unnecessary to consider these. The diligent reader may have noticed that the contextual backward function evaluation of the previous section may generate entry assertions ϕ_E which do not satisfy the property (3.7). The wff ϕ_E is essentially ϕ_B from which some disjuncts are deleted.

To illustrate this dramatically, consider the one-line program segment $S \equiv$ exchange x_i with x_j, the exit assertion ψ_S being sorted $(1,n)$. The ϕ_B generated by true backward substitution is the equivalent of that shown in Figure 3.10a. This ϕ_B does imply the property (3.7). (For readability we have not written ϕ in disjunctive normal form.) However, the backward function evaluation in the context of $1 \leq i < j \leq n$ yields ϕ_E shown in Figure 3.10b. As can be seen, ϕ_E is much simpler than ϕ_B, whose generation does not depend on the context of the given entry assertion.

Theorem 5 (Validity of contextual backward function evaluation)

For a straightline program segment S,

$$\{\phi \mid S \mid \psi\} \quad \text{iff} \quad \phi^* \models \phi_E$$

Proof Without loss of generality, we assume that the given entry assertion is ϕ^*, the enriched version of ϕ. Thus, we wish to prove

$$\{\phi^* \mid S \mid \psi\} \quad \text{iff} \quad \phi^* \models \phi_E. \tag{3.9}$$

We actually prove a slightly stronger version than (3.9), namely,

$$\{\phi^* \mid S \mid \psi\} \quad \text{iff for every disjunct } \phi_i^* \text{ of } \phi^*,\ \phi_i^* \models \phi_{Ei}$$

$*$?

$S \equiv$ exchange x_i with x_j

$*$ sorted(1,n)

$\phi_B \equiv \phi_E$ with context 'true'

$\equiv 1 \le i \le j \le n$ and sorted(1,i-1) $\le x_j \le$ sorted(i+1,j-1) $\le x_i \le$ sorted(j+1,n)

or $1 \le j < i \le n$ and sorted(1,j-1) $\le x_i \le$ sorted(j+1,i-1) $\le x_j \le$ sorted(i+1,n)

or $1 \le i \le n < j$ and sorted(1,i-1) $\le x_j \le$ sorted(i+1,n)

or $j < 1 \le i \le n$ and sorted(1,i-1) $\le x_j \le$ sorted(i+1,n)

or $1 \le j \le n < i$ and sorted(1,j-1) $\le x_i \le$ sorted(j+1,n)

or $i < 1 \le j \le n$ and sorted(1,j-1) $\le x_i \le$ sorted(j+1,n)

or sorted(1,n) and (i < 1

or i > n)

and (j < 1

or j > n)

(a) ϕ_B: Context-Free Backward Substitution (Simplified)

ϕ_E in the context of $1 \le i \le j \le n$

$\equiv$ sorted(1,i-1) $\le x_j \le$ sorted(i-1,j-1) $\le x_i \le$ sorted(j+1,n)

(b) ϕ_E: Contextual Backward Substitution (Simplified)

Figure 3.10 Contextual and Context-Free Backward Substitutions

where ϕ_{Ei} is the entry assertion of S obtained by evaluating the backward functions in the context of L_i, the linear ordering of boundaries defined by the (enriched) disjunct ϕ_i^*. That $\phi_i^* \models \phi_{Ei}$, say for i = 1, is shown by proving

$$\phi_B \text{ and } L_1 \models \neg\phi_{E1} \text{ and } L_1$$

Thus, ϕ_E is ϕ_{E1} or ϕ_{E2} or . . . .

ϕ_B and $L_1 \models \phi_{E1}$ and L_1:

Suppose ϕ_B and $L_1 \not\models \phi_{E1}$. Let M be a model for ϕ_B and L_1, such that $\not\models_M \phi_{E1}$. Then ϕ_{E1} and L_1 is not true in M. By Lemma 3 of Section 3.3.2, if $\not\models_M \phi_{E1}$ and L_1 then $\not\models_{M'} \psi$, where M' is the result of the execution of S on M. Since M is a model for ϕ_B, this contradicts property (3.7).

ϕ_{E1} and $L_1 \models \phi_B$ and L_1:

Suppose ϕ_{E1} and L_1 is true in M, and $\not\models_M \phi_B$. By property (3.7) $\not\models_M \psi$. But by Lemma 3, if $\models_M \phi_{E1}$ and L_1 then $\models_M \psi$, a contradiction. ∎

The advantage in generating ϕ_E's rather than ϕ_B should be obvious. The assertion ϕ_B will have as many disjuncts as there are linear orderings of the boundaries collected from S and ψ_S. Several of these linear orderings are of no concern to us, since we only need to prove that execution of S on an input staisfying ϕ results in ψ. We do not care what S does on a linear ordering of boundaries contradicting the partial order specified by ϕ.

Since contextual backward function evaluation is valid, the extended theorem prover is a decision procedure for $\Omega \models \omega$, where ω and Ω are augmented wffs.

3.5 Counterexample Generation

Whenever the theorem prover determines that $\Omega \not\models \omega$ it is possible, in this system, to construct a model M for Ω such that ω is false in M. However, it should be realized that M may not be a "counterexample to the program." This is because even though $\{\phi|P|\psi\}$, the loop invariants given may not be strong enough to prove all the lemmas generated. Counterexamples will, hopefully, provide clues for strengthening the loop invariants.

Suppose $\Omega \not\models \omega$. Then there must exist (see Algorithm 2: prove-theorem) a linear ordering L, ptr graph π and key graph α of a disjunct Ω_1 of Ω such that

$$\pi \text{ and } L \text{ and } \alpha \not\models \omega.$$

Let $\omega^{\#}$ be the partitioned version of ω in the context of L,

$$\omega \text{ and } L \models \dashv \omega^{\#} \text{ and } L$$

$$\omega^{\#} \equiv \omega_1^{\#} \text{ or } \omega_2^{\#} \text{ or } \ldots \text{ or } \omega_c^{\#}.$$

The last call (from either Algorithm 2 or 4) of Algorithm 3: provelemma gives a satisfiable disjunct Ω_{11}, and $\omega' \equiv \omega_c^{\#}$ such that

$$\Omega_{11}^{*} \not\models \omega_{c}^{\#}$$

and for $1 \leq i < c$,

$$\Omega_{11}^{*} \text{ and } \omega_{i}^{\#} \text{ is unsatisfiable.}$$

Thus, a model M for Ω_{11}^{*} such that $\not\models_{M} \omega_{c}^{\#}$ is a counterexample to (π and L and $\alpha \models \omega$) and hence to $\Omega = \omega$. Since $\Omega_{11}^{*} \not\models \omega_{c}^{\#}$, there must be a predicate p in $\omega_{c}^{\#}$ such that $\Omega_{11}^{*} \not\models p$ (see Algorithm 4). The disjunct Ω_{11}^{*} and not p is satisfiable, and a model can be constructed as in Section 3.2.1 for the transitive closures of Ω_{11}^{*} and not p.

4. GENERALITY

In the last two chapters, we have seen the successful application of inference rules about partitioning, closure and local emplication in the verification of programs written and asserted in our languages. Though these vital inference rules are developed here as the result of severe constraints imposed primarily by the assertion language, they do apply to a wider class of programs manipulating data structures. We now give several examples to support this contention.

4.1 Constraints of the Present Verification System

The verification system was designed with the specific goal of being usable in SORTLAB to verify the correctness of student programs for sorting an array. Severe constraints were imposed on the programming and assertion languages both to limit the class of programs to sorting-type problems and to obtain a system that is usable in a practical situation. Not all these constraints are technically necessary for making the theorem prover a decision procedure, though they have value pedagogically.

For example, the verifier can be enhanced quite easily to permit many arrays, temporary variables, ptr expressions like j + 8, and predicates like $\underline{\text{array}}\ (s,t) - 3 \leq \underline{\text{array}}\ (u,v)$, which means

$$\underline{\text{array}}\ (s,t) - 3 \leq \underline{\text{array}}\ (u,v) \equiv \forall_i \forall_j\ (s \leq i \leq t\ \underline{\text{and}}\ u \leq j \leq v \rightarrow x_i - 3 \leq x_j).$$

However, if arbitrary assignments to array elements are allowed, it is

not clear how the verifier can be extended to prove the key-preserving property of solving algorithms.

It is not possible to characterize the class of programs provable in this system except as those programs that can be written in our programming language and for which sufficiently strong assertions can be made in our assertion language. Theoretically speaking, all computable functions are programmable in the programming language. However, for most computable functions strong enough assertions do not exist in our assertion language that permit a proof that the corresponding program computes the function. Thus, e.g., heap sort and several merging programs can be written in the programming language, but strong enough assertions to prove that these programs also sort do not exist in our assertion language.

4.2 Partitioning

Several properties on a data structure can be expressed as properties on its substructures, and by interrelationships among these components. For example,

sorted (s,t) iff $s \geq t$ or (for all u, $s \leq u < t$
sorted (s,u) $\leq$ sorted (u+1,t))

avl-tree iff empty-tree (r) or
avl-tree (left (r)) and avl-tree (right (r))
and $-1 \leq$ height (left (r)) -height (right (r)) ≤ 1

A typical verification condition $\Omega \models \omega$ of a program aiming to produce such a property on a data structure is of the following kind: the conclusion

ω refers to larger parts of a data object having the property, while the premise Ω refers to smaller parts of the data object which have the same (or similar) property and contains certain interrelationships between these parts. Proving $\Omega \models \omega$ becomes much simpler in such cases if both Ω and ω are expressed in terms of a set of common parts of the data object. Partitioning is a technique which decomposes the data object into small enough components so that every segment of data structure referred to in Ω or ω is a union of some of these components.

4.3 Closure and Local Implication

Much of the inefficiency in general theorem provers can be traced to their inability to choose appropriately those predicates of the premise which would imply a certain conclusion. The rule of local implication completely avoids this problem by specifying the predicate of the premise that determines if a given predicate of the conclusion follows from the premise. It should be noted that the rule of local implication is valid only when the ptr and key graphs are transitively closed.

A rule of local implication can trivially be formulated in any deductive system if all possible inferences from the given premises are collected as the closure of the premise. However, this may not be practical either because it takes a long time or because the closure is not finite. We therefore seek inference rules yielding only finitely many inferences from given premises and obtain the closure of such rules. In the context of proving lemmas about partitionable properties

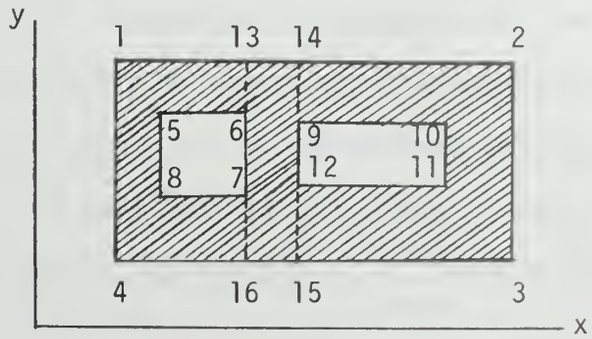
on data structures, it is generally possible to obtain this closure rapidly, and to invent appropriate rules of local implication.

4.4 Examples

Several examples from the literature are used in this section to support our contention that the techniques developed for SORTLAB are in fact applicable to a wider class of programs. The treatment of these examples is necessarily brief; we only indicate how a relevant partition may be constructed. We also assume, without further ado, that appropriate extensions are made to the programming and assertion languages where necessary.

4.4.1 A Geometric Example

Consider finite plane maps which can be described using rectangles with one side parallel to the x-axis, and the operations union (+), intersection (·) and negation (¬). Thus A + B is the map covered by the rectangle A or B, A.B represents the map common to both A and B, and ¬A represents the map not covered by A. The shaded map shown below can be described by several expressions.



For example,

(1-2-3-4)•¬(5-6-7-8)•¬(9-10-11-12)

(1-14-15-4)•¬(5-6-7-8) + (13-2-3-16)•¬(9-10-11-12)

(1-2-3-4)•¬(5-10-11-8) + (6-9-12-7)

The problem we wish to consider is: given two expressions E_1 and E_2, decide if E_1 and E_2 are describing the same map. If the coordinates of all points referred to in E_1 and E_2 are constants, the problem is trivial. But, if the points are arithmetic expressions (with plus, minus only) of free variables and constants, the problem can be answered by decomposing the maps described by E_1 and E_2 as follows.

Let the rectangle A contain a corner p of another rectangle B. Then, p splits A into four smaller rectangles A_1, A_2, A_3 and A_4 as shown below. Repeat this process until none of the partitioned rectangles

contain corners of other rectangles. Clearly, each original rectangle is a union of some of these partioned rectangles. If we now impose a linear ordering on these partitioned rectangles (e.g., A precedes B if the coordinates of the left-top corner of A are (x_1,y_1) and that of B are (x_2,y_2) such that either $x_1 < x_2$ or $x_1 = x_2$ and $y_1 < y_2$) the original expressions E_1 and E_2 can be rewritten in a canonical form now and E_1 will be equivalent to E_2 if their partitioned expressions are identical.

4.4.2 Simple Array Examples

All the verification conditions of the two examples given in this section can be proven by partitioning the array as described in Section 3.1.3.4.

4.4.2.1 Binary Search

The example given in Algorithm 7 is a classical binary search algorithm. The proof that the algorithm searches correctly a sorted array x(m...n) for an element z does not depend on the index k being equal to (i+j) div 2; this particular choice of k only makes the algorithm more efficient ($O(\log_2(m-n))$). For the algorithm to search properly it is sufficient that the function f be such that whenever $i < j$, $i \leq f(i,j) < j$. The verification condition for the loop is:

sorted (m,n) and $i \leq k < j$ and
(z isin-array (i,j) or z notin-array (m,n))
$\models$
sorted (mn,) and $i \leq k$ and $x_k \geq z$ and
(z isin-array (i,k) or z notin-array (m,n))
or
sorted (m,n) and $k + 1 \leq j$ and $x_k < z$ and
(z isin-array (k + 1, j) or z notin-array (m,n))

The predicate notin-array is the negation of isin-array, where

$$z \text{ isin-array } (s,t) \equiv \left| \begin{array}{l} s = t \text{ and } x_s = z \text{ or} \\ (\text{for some } u \text{ such that } s \leq u < t \\ z \text{ isin-array } (s,u) \text{ or } z \text{ isin-array } (u+1,t)) \end{array} \right.$$

```
*sorted (m,n)
i ← m; j ← n
while i < j do
        k ← f(i,j)
        if x_k < z
        then i ← k + 1
        else j ← k
        endif
  * i ≤ j and (z isin-array (i,j) or z  notin-array (m,n))
endwhile                                and sorted (m,n)
found ← (x_i = z)
* (found ↔ z isin-array (m,n))
```

Algorithm 7. Classical Binary Search

4.4.2.2 Dutch National Flag Problem

The problem is to rearrange the elements of an array x which are those-valued viz., either red, white or blue, into contiguous red-, white- and blue-colored segments from the low end to high end respectively. [Dijkstra 1976]. A solution to the problem is given here as Algorithm 8. The predicates red, white, blue or array segments are defined as follows:

$$
\begin{aligned}
c(s,t) \equiv (s < t \ \underline{\text{and}} \ & \text{for all } u \text{ such that } s \leq u < t \\
& c(s,u) \ \underline{\text{and}} \ c(u+1,t) \\
\underline{\text{or}} \ & s = t \ \underline{\text{and}} \ \text{color}(s) = c \\
\underline{\text{or}} \ & s > t)
\end{aligned}
$$

where c is to be substituted by red, white, or blue. The backward function evaluation, and partitioning technique of Chapter 3 are adequate to prove the partial correctness of this algorithm.

4.4.3 Heap Sort

Algorithm 9 [Floyd 1964] imposes the structure of a binary tree on the array to sort its elements. We formulate the sift-up algorithm recursively; an iterative version of this algorithm is not provable using our partitioning technique (see Section 4.5). The predicates ordt, $x \geq$ tree $(\cdot,\cdot)$ are defined below:

```
r ← 1; w ← 1; b ← n
while w ≤ b do
      cases color (w) of
         white:  w ← w + 1
         red:    (exchange t_w with t_r;
                  r ← r + 1; w ← w + 1)
         blue:   (exchange x_w with x_b;
                  b ← b - 1)
      end cases
* red (1,r-1) and white (r,w-1) and blue (b+1,n) and
                                  1 ≤ r ≤ w ≤ b ) ≤ n + 1
endwhile
* red (1,r-1) and while (r,w-1) and blue (w,n)
             and 1 ≤ r ≤ w = b + 1 ≤ n + 1
```

Algorithm 8. Dutch National Flag

$x_u \geq \underline{tree}\ (s,t) \equiv (s \leq t\ \underline{and}\ x_u \geq x_s\ \underline{and}\ x_u \geq \underline{tree}\ (2s,t)$
$\underline{and}\ x_u \geq \underline{tree}\ (2s+1,t)$
$\underline{or}\ s > t)$

$\underline{ordt}\ (s,t) \equiv (s < t\ \underline{and}\ x_s \geq \underline{ordt}\ (2s,t)$
$\underline{and}\ x_s \geq \underline{ordt}\ (2s+1,t)$
$\underline{or}\ s \geq t)$

$x_u \geq \underline{ordt}\ (s,t) \equiv (x_u \geq \underline{tree}\ (s,t)\ \underline{and}\ \underline{ordt}\ (s,t))$

$\underline{heap}\ (s,t) \equiv (s < t\ \underline{and}\ \underline{heap}\ (s+1,t)\ \underline{and}\ \underline{ordt}\ (s,t)$
$\underline{or}\ s \geq t)$

Since our interest here is to demonstrate the applicability of the principle of partitioning, we shall take the liberty of simplifying the verification conditions. A crucial verification condition of siftup-procedure is:

$\{j = 2i < n\ \underline{and}\ x_j < x_i\ \underline{and}\ x_{j+1} \leq x_i\ \underline{and}$
$\underline{ordt}\ (2j,n)\ \underline{and}\ \underline{ordt}\ (2j+1,n)\ \underline{and}$
$x_i \geq \underline{tree}\ (j,n)\ \underline{and}\ x_i \geq \underline{ordt}\ (j+1,n)$
$|\underline{call}\ \ \text{siftup}\ (j,n)|$
$\underline{ordt}\ (i,n)\},$ (4.1)

```
procedure siftup (i,n)
  * ordt (wi,n) and ordt (2i + 1, n)
  j ← 2 * i
  if j ≤ n then
     if j < n then
        if x_j < x_j+1 then j ← j + 1 endif
     endif
     if x_i < x_j then
        exchange x_i with x_j;
        * ordt (2j,n) and ordt (2j + 1, n) and x_i ≥ tree (j,n)
                                              and x_i ≥ ordt (j + 1, n
        call siftup (j,n)
     endif
  endif
  * ordt (i,n)
endproc
```

Algorithm 9(a). Recursive Siftup Algorithm

```
procedure heapsort (n)
   for i   n div 2 downto 2 do
       call siftup (i,n)
       * heap (i,n) and 2 ≤ i ≤ n div 2
   endfor
for i ← n downto 2 do
       call siftup (1,i);
       exchange x1 with xi
       * heap (2, i-1) and array (1, i-1) ≤ sorted (i,n) and ≤ i ≤ n
   endfor
 * sorted (1,n)
endproc
```

Algorithm 9(b). Heap Sort

<u>assuming</u> that siftup does not change the order of elements in any <u>tree</u> (s,n) unless the tree is a subtree of <u>tree</u> (i,n). The Lemma (4.1), therefore, reduces to:

$$
\begin{aligned}
& j = 2i < n \ \underline{\text{and}} \\
& \underline{\text{ordt}}\ (2j,n)\ \underline{\text{and}}\ \underline{\text{ordt}}\ (2j+1,n)\ \underline{\text{and}} \\
& x_i \geq \underline{\text{tree}}\ (j,n)\ \underline{\text{and}}\ x_i \geq \underline{\text{ordt}}\ (j+1,n)\ \underline{\text{and}} \\
& \underline{\text{ordt}}\ (j,n) \\
& \models \\
& \underline{\text{ordt}}\ (i,n)
\end{aligned}
\qquad (4.2)
$$

where <u>call</u> siftup (j,n) has added <u>ordt</u> (j,n). The relevant partition of the "array" is <u>not</u> decomposing into contiguous array segments but to decompose the <u>tree</u> (i,n) into its two subtrees <u>tree</u> (2i,n) and <u>tree</u> (2i+1,n) and the root x_i. The proof of (4.2) requires consideration of three cases: $2j > n$, $2j = n$, and $2j < n$. To demonstrate the use of a partition of the above type, consider the most interesting case $2j < n$. We can rewrite (4.2) as:

$$
\begin{aligned}
& j = 2i < n \ \underline{\text{and}}\ 2j < n \ \underline{\text{and}} \\
& \underline{\text{ordt}}\ (2j,n)\ \underline{\text{and}}\ \underline{\text{ordt}}\ (2j+1,n)\ \underline{\text{and}} \\
& x_i \geq \underline{\text{tree}}\ (2j,n)\ \underline{\text{and}}\ x_i \geq \underline{\text{tree}}\ (2j+1,n)\ \underline{\text{and}}\ x_i \geq x_j\ \underline{\text{and}} \\
& x_i \geq \underline{\text{ordt}}\ (j+1,n)\ \underline{\text{and}} \\
& x_j \geq \underline{\text{ordt}}\ (2j,n)\ \underline{\text{and}}\ x_j \geq \underline{\text{ordt}}\ (2j+1,n) \\
& \models \\
& x_i \geq x_{2i}\ \underline{\text{and}}\ x_i \geq \underline{\text{ordt}}\ (4i,n)\ \underline{\text{and}}\ x_i \geq \underline{\text{ordt}}\ (4i+1,n)\ \underline{\text{and}} \\
& x_{2i} \geq \underline{\text{ordt}}\ (4i,n)\ \underline{\text{and}}\ x_{2i} \geq \underline{\text{ordt}}\ (4i+1,n)\ \text{and} \\
& x_i \geq \underline{\text{ordt}}\ (2i+1,n)
\end{aligned}
\qquad (4.3)
$$

As can be seen, the conclusion follows from the premise if 2i is substituted for j.

The verification conditions for the two for-loops of heapsort (Algorithm 9(b)) require even more complex partitioning: a decomposition into subtrees as well as into array segments of one-element. However, an iterative version of the siftup-algorithm does not yield to such a decomposition of the heap, and hence is not provable by our techniques.

4.4.4 A List Moving Algorithm

Algorithm 10 [Reingold 1973, Wagner 1974] moves all nodes of a list structure accessible from a root to a new contiguous set of nodes. We outline a proof of the fact that what is copied by the algorithm is isomorphic to the original list structure composed of all, and only, those nodes accessible from the root. For convenience in this proof, we have introduced the tables copyof [·] and origof [·], and boolean flags copied [·]. The original node, origof [q], of the newly copied node q is not required by the algorithm itself; the tables copied [·] and copyof [·] may be overlapped with the left [·] fields of the original nodes (see Wagner 1974). The predicates in the loop invariant are defined below:

$$
\begin{aligned}
\underline{\text{isocopy}}\ (q) \equiv\ & (q = 0\ \underline{\text{or}} \\
& \underline{\text{isocopy}}\ (q\text{-}1)\ \underline{\text{and}}\ \text{data}\ [q] = \text{data}\ [q_0]\ \underline{\text{and}} \\
& \text{right}\ [q] = \underline{\text{copyof}}\ [\text{right}\ [q_0]]\ \underline{\text{and}} \\
& \text{left}\ [q] - \underline{\text{copyof}}\ [\text{left}]q_0]])
\end{aligned}
$$

```
procedure movelist (root)
   p ← 0; q ← 0; ℓ ← root
   call copy (ℓ)
   while q   p do
      q ← q + 1
      call copy (left [q])
      call copy (right [q])
      * isocopy (q) and q to p and p from q and dupe (q,p)
   endwhile
   * isocopy (p) and p to p and p from p
endproc


procedure copy (var x)
   if x ≠ nil then
      if not copied [x]
      then p ← p + 1;
           node [p] ← node [x];
           copied [x] ← true;
           copyof [x] ← p;
           origof [p] ← x
      endif
      x ← copyof [x]
   endif
endproc
```

Algorithm 10 A List Moving Algorithm

where q_o = origof [q]. (The nodes 1 through q constitute an isomorphic copy of a substructure of the original list.)

q to p

≡ (q = 0 or

q-1 to p and left [q) ≤ p and right [q] ≤ p or

q-1 to p-1 and (right [q] ≤ p-1 and left [q] = p or

right [q] = p and left [q] ≤ p-1) or

q-1 to p-2 and left [q] = p-1 and right [q] = p)

p from q

≡ (q = 0 or p from q-1 or

p-1 from q-1 and (p = left [q] or p = right [q]) or

p-2 from q-1 and p-1 = left [q] and p = right [q])

(q to p means that all nodes reachable from q using right-left links are included in 1 . . . p. Similarly, p from q denotes the converse, i.e., all nodes included in 1 . . . p are reachable from nodes in 1 . . . q via the right-left links.)

dupe (s,t) ≡ (s > t or s = t and node [s] = node [origof[s]] or

for some u, s ≤ u < t and dupe[s,u] and

dupe [u+1,t])

(Nodes from s to t are exact copies of their original nodes.)

The partition of the copied list structure as indicated by the above definitions of the predicates readily gives a proof of various verification conditions of the list moving algorithm.

4.5 On the Applicability of Partitioning

As we have seen in the examples of the proceeding section, a class of programs that typically have loops (recursive calls) operate on their data objects building up the desired property iteratively (recursively). Two general approaches are discernible in the iterative build-up of properties:

A1. The data structure having a desired property P is gradually built-up. If D is a segment of the data object having property P, we find δD, an incremental part from the remaining part of the data object. The composite segment $D + \delta D$ is manipulated so that $D + \delta D$ has the property P. Repeat the process until all of the data object has the property P [Misra 1976].

A2. The desired property P on a data object is gradually built-up. If D has a property Q, we manipulate D so that it now has property Q' which is "closer" to P than Q was.

The examples of Section 4.4.2 and 4.4.4 belong to class A1. Partitioning seems applicable to all such programs. It is, of course, possible to describe an algorithm belonging to class A1 in terms of A2. A bubble sorting algorithm can be thought of as converting an array that is less-sorted to an increasingly-sorted array; however, the algorithm is best put in class A1. On the other hand, there are algorithms belonging to class A2 which it will be very difficult to describe in terms of A1. A nonrecursive sift-up algorithm of heap sort (see, Floyd 1964 and Section 4.3) descends the tree confining the undesirable property

that some tree is not ordered (ordt) to smaller and smaller trees. This algorithm clearly belong to Class A2.

Thus, for partitioning to be applicable, it seems necessary that the following requirements be satisfied:

R1. The data structures used must have disjoint components. (Thus circular lists, "trees" with shared structures do not satisfy this requirement, while stacks, queues, linear lists, trees, tables do.)

R2. It should be possible to describe the property P on data object D equivalently in terms of the same property P on components of D obtained by a finite decomposition, and possibly some interrelationships among the components. (Properties like A is a permutation of A_0, are not thus partitionable, while those like T is an AVL-tree, array A is sorted, or array A is a heap are.)

R3. The property P being sought should be built-up by the algorithm using the approach A1.

When the desired property P, and data object D satisfy requirements R1 and R2, it is generally possible to write programs that satisfy R3. Thus, the applicability of partitioning depends not only on the intrinsic properties of the data structure, and the property P, but also on how P is built-up.

5. SORTLAB

The verification system described in Chapters 2 and 3 is at the heart of a programming laboratory, called SORTLAB, which assists the student-programmer in producing correct sorting algorithms from basic ideas of these algorithms. SORTLAB consists of a program editor, an interpreter, the program verifier described earlier and a counter-example generator. These are implemented on the PLATO interactive system as a "lesson." This lesson is a part of the Automated Computer Science Education System (ACSES) developed by the Department of Computer Science of the University of Illinois.

This chapter describes SORTLAB, its use and its implementation. Sections 5.1 and 5.2 provide a context in which the performance of SORTLAB should be evaluated.

5.1 PLATO

The PLATO IV interactive system [Alpert and Bitzer 1970] is designed to support more than 500 users logged-in on the plasma-panel graphic terminals. The users can be divided into "authors" who write teaching-programs ("lessons"), and "students" who execute these lessons at their own pace. It is expected that a user limit CPU usage to 2 milliseconds/clock-second; any attempted over-use will be reduced to this level by offering fewer time-slices.

Each student-user has a data segment of 1,650 60-bit words. A lesson is assigned a data space of 1,500 words in the central memory, and it can access these 1,500 words and the first 150 words of student

data segment. The 1,500-word space must be loaded (and unloaded) with the contents of the remaining 1,500 words of student data segment or of a segment of extended core storage containing information that is common to all users executing the lesson. Thus any lesson using more than 150 words of data must explicitly control this "paging."

The single most annoying factor in the use of the PLATO system for program development is TUTOR, the only programming language available to authors, in which the lessons are to be written. (For a short introduction, see Popular Computing 1975; a detailed, and a slightly outdated description may be found in [Sherwood 1975].) TUTOR is a high-level language with an assembly-language-like format. It contains several machine-dependent data manipulative statements with such niceties as nested assignment statements and generalized versions of the computed-goto and do-loop statements of FORTRAN. Procedure blocks may be defined, but there are no local variables. Each variable name must be assigned an address by the programmer. Several variables with small values may be assigned to different segments of the same 60-bit word. In addition to these features, there are several statements that are useful in judging the students response. The run-time system of TUTOR permits nested procedure calls (recursive or not) at most 10 levels deep. Most lessons written for PLATO have a simple structure; for these programs, lack of control structures, local variables, etc. are not serious impediments. Typically, such lessons also use little CPU-time. Most students find it pleasant to "read" such lessons because of the near-instantaneous response and excellent graphics. Any unpleasantness is usually attributable to the author's style of writing his lesson.

5.2 ACSES

The Department of Computer Science of the University of Illinois has developed on PLATO an Automated Computer Science Education System [Nievergelt 1975] for beginning students in computer science. It consists of a large body of lessons, a GUIDE information retrieval and management system [Eland 1975] and an interactive programming system [Wilcox 1973]. The GUIDE may be used by a student to find out about his records or to choose a lesson of interest. The programming system supports several languages with excellent error diagnostics. The body of lessons largely consists of conventional Computer Assisted Instruction lessons about various aspects of computer science. Among this collection are two lessons which incorporate novel concepts of artificial intelligence and program proving adapted to run on limited computer resources: PATTIE [Danielson 1975], to tutor students in top-down program design; and SORTLAB, to be presented in the next section.

53. SORTLAB--A Programming Laboratory

SORTLAB concerns itself with the implementation of certain sorting algorithms. It provides a "laboratory" wherein a student can perform programming "experiments" using the various equipment provided. It does not actively suggest what ways should be used in implementing an algorithm, but focuses the student's attention on the correctness of his program by providing such tools as specially-designed, and easy-to-learn mini programming language, an excellent program editor, a program verifier, a counter-example generator, and an interpreter for his programs.

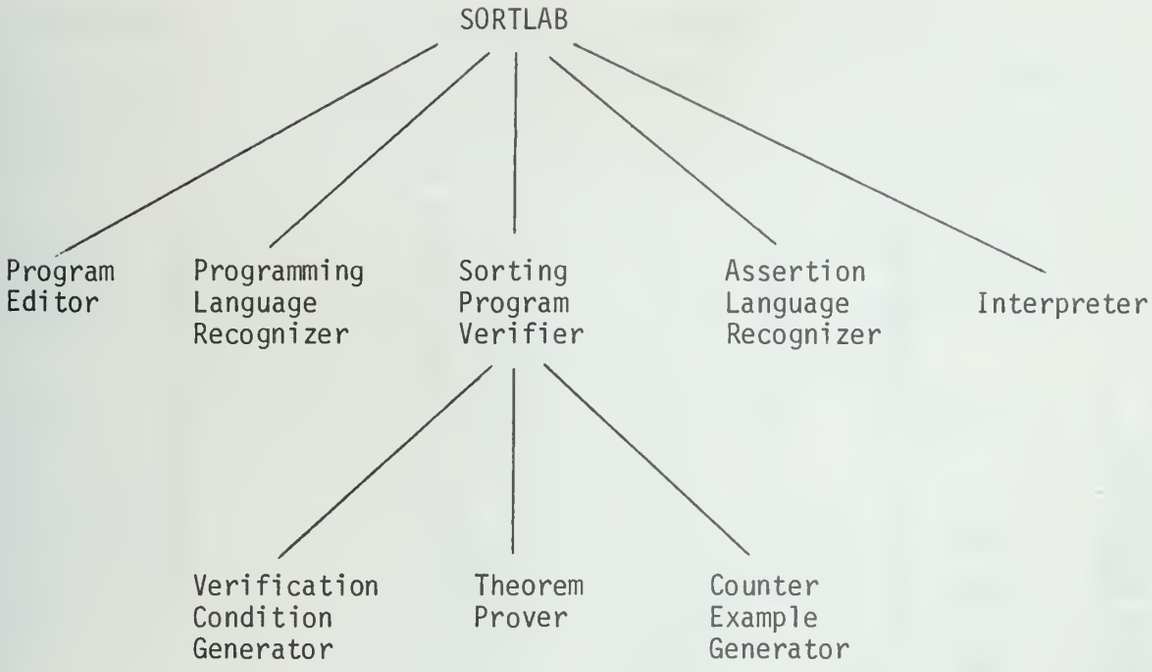
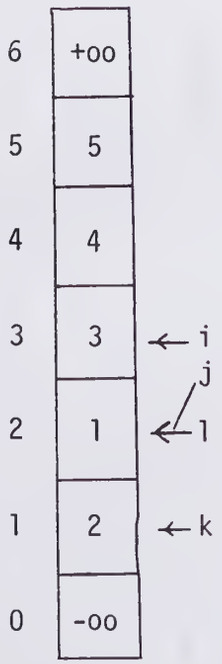


Figure 5.1 Components of SORTLAB

5.3.1 Programming and Assertion Languages

The languages are so chosen that while it is convenient and natural to express several sorting algorithms, writing other programs is not easy. The particular choice of basic operations in the programming language, and predicates in the assertion language is strongly influenced by decidability considerations (see Section 2.2).

A program example is given in Figure 5.2. The syntax of the languages is specified in Figures 5.3 and 5.4. The assertion language semantics is specified in Section 3.1.1. The ptr assignment, while, if and call statements have the conventional meaning. The semantics of other statements of the programming language is explained in the examples below.




```
1   procedure part (k;l)*(j)
*   XK-1 ≤ A(K;L) ≤ XL+1 & K < L
2     i ← k+1
3     j ← l
4     while i ≤ j do
5       while xk > xi do
6         i ← i+1
*           XK-1 ≤ A(K+1;I-1) ≤ XK ≤ A(J+1;L) ≤ ....
7       endwhile
8       while xk < xj  do
9         j ← j-1
*         XK-1 ≤ A(K+1;I-1) ≤ XK < A(J+1;L) ≤ ....
10      endwhile
11      if i ≤ j then
12        exchange xi with xj
13        i ← i+1
14        j ← j-1
15      else
16      endif
quik    *       XK-1 ≤ A(K+1;I-1) ≤ XK ≤ A(J+1;L) ≤ ....
quik    17    endwhile
sort    18    exchange xk with xj
*   XK-1 ≤ A(K;J-1) ≤ XJ ≤ A(J+1;L) ≤ XL+1 & K
19  endproc
```

```
::: executing   6 *   :::
6*  assertion is true!
array display on                          execution can be resumed
```

Figure 5.2: A Display Page During Execution

```
<procedure>              ::= procedure <identifier><input par>
                             <optional out par exp><stmt list> endproc
<stmt list>              ::= {<stmt>}*
<stmt>                   ::= <ptr-assign>|<exchange>|<insert>|<call>|
                             <while>|<scan>|<if>
<while>                  ::= while <bool exp> do <stmt list> endwhile
<scan>                   ::= scan <updn with><ptr var> from <ptr exp>
                             to <ptr exp><stmt list> endscan
<if>                     ::= if <bool exp> then <stmt list> else
                             <stmt list> endif
<ptr-assign>             ::= <ptr var> ← <ptr exp>
<exchange>               ::= exchange x <ptr exp> with x <ptr exp>
<insert>                 ::= insert x <ptr exp> below x <ptr exp>
<call>                   ::= call <proc identifier> (<ptr exp>, <ptr exp>)
                             <optional out par var>
<optional out par var>   ::= <empty>|*(<ptr var>{,<ptr var>}*)
<input par>              ::= (<ptr var>, <ptr var>)
<optional out par exp>   ::= <empty>|*(<ptr exp>{,<ptr exp>}*)
<bool exp>               ::= <pl-disjunct> {or <pl-disjunct>}*
<pl-disjunct>            ::= <pl-predicate> {and <pl-predicate>}*
<pl-predicate>           ::= <ptr pred>|<key pred>
<ptr pred>               ::= <ptr exp><nerel><ptr exp>
<key pred>               ::= x <ptr exp><nerel>x<ptr exp>
<nerel>                  ::= <rel>|≠
<rel>                    ::= < | ≤ | = | ≥ | >
<updn with>              ::= up with|down with
<ptr exp>                ::= 0|1|2|<ptr var>|<ptr var> ± 1
<ptr var>                ::= i|j|k|l|m|n
```

Figure 5.3. Syntax of the Programming Language

```
<assertion>          ::= <disjunct> {or <disjunct>}*
<disjunct>           ::= <predicate> {and <predicate>}*
<predicate>          ::= <ptr predicate>|<array predicate>
<ptr predicate>      ::= <ptr exp>{<rel><ptr exp>}+
<array predicate>    ::= sorted <segment def>|<segment>
                         {<rel><segment>}+
<segment>            ::= array <segment def>|sorted
                         <segment def>|x <ptr exp>
<segment def>        ::= (<lower boundary>, <upper boundary>)
<lower boundary>     ::= <boundary>
<upper boundary>     ::= <boundary>
<boundary>           ::= <ptr exp>
```

Figure 5.4. Syntax of the Assertion Language

The statement

```
scan up with i from j + 1 to k - 1
    <body>
endscan
```

is equivalent to

```
i ← j + 1
while i ≤ k - 1 do
    <body>
    i ← i + 1
endwhile
```

The loop variable i of the scan statement is not considered unmodifiable by the body.

The statement "insert xi below xj" is equivalent to the following abstract program:

```
t ← x_i; p ← i
if i ≤ j then
   while p ≤ j - 2 do x_p ← x_{p+1}; p ← p + 1 endwhile
   {circular up shift}
else while p ≤ j + 1 do x_p ← x_{p-1}; p ← p - 1 endwhile
   {circular down shift}
endif
x_p ← t
```

A program, in SORTLAB, is a collection of procedures and it always includes the main procedure "sort." All procedures are external and may be recursive. The array x is global to all procedures; indices are always local. Thus, the only way a procedure may receive an index value is by receiving it as a (value) parameter.

Notice that apart from the array to be sorted x, and ptr variables, no temporary variables are provided. Two padding elements x_0, and x_{n+1} are predefined to be $-\infty$ and $+\infty$ respectively; these may be used as sentinels. Thus, the entry and exit assertions of main procedures sort (n) are:

$n \geq 1$ and $x_0 <$ array $(1,n) < x_{n+1}$

sorted $(1,n)$

5.3.2 Language Recognizers

The tokens of the programming and assertion languages are so chosen that (except for if, insert, and i ← . . .) they can be recognized by their first character. As soon as the first character of the token is typed, the statement is completed as far as possible and is displayed. An illegal key-press causes it to be flashed and is ignored. Thus, in writing the following statements only the underlined keys need be pressed:

scan down with i from n to 2

endscan

exchange xi-1 with xj+1

5.3.3 Program Editor

Each procedure constitutes a "display page," and these may be

selected by typing in the name of the procedure. A statement is inserted by first giving a line number to it and then writing the statement. An assertion is given as the exit assertion of a statement; the assertion is displayed at the end of the statement. Thus, the line labeled 16* in Figure 5.2 is the exit assertion of the if-statement at line 11. It is also the loop invariant of the while-loop at line 4. Any sequence of statements can be deleted and, if so desired, saved. A segment from among several of such saved program segments may later be inserted into a procedure.

Compound statements like the while-statement are written in two steps: first, the while-envelope with its corresponding endwhile and without a body is written. At a later time, the body is formed either as a sequence of new statements, or by inserting a saved program segment. Thus, a number of simple, but common, errors, like unmatched end-brackets of statements, unintentional nesting of bodies because of a missing begin, end, or semicolon, do not arise. Further, structural changes of a procedure do not require reparsing. Every structural change results in a new page displaying the updated version, with automatic indentation, of the procedure.

A number of ideas incorpoarted into this editor are originally due to [Hansen 1971].

5.3.4 Interpreter

The interpreter can execute any program written in the programming language. The assertions are also executed, and their truth value at run-time is indicated. It is possible to execute the program

in various modes, including step-by-step node. During execution, the contents of the array being sorted is dynamically displayed along with the location of various indices (Figure 5.2). Only the currently active procedure are displayed; as each new procedure is entered, that procedure is displayed. An invocation trace is also displayed.

The interpreter carefully checks for all possible violations of the assumptions made by the verification system: Each procedure is assumed to permute only the elements of the array segment between the two imput parameters of the procedure (1 is an "implicit" input parameter of procedure sort; this prevents it from becoming a recursive procedure since each call statement must have two input (actual) parameters!). The values of all index variables should be between 0 and n + 1 where n is the size of the array; once an index variable has a value outside this range, it is not possible for that variable to have a legal value.

5.3.5 Sorting Program Verifier

The student requests that his program be verified when he has completed writing it. The verifier then proceeds to verify his program provided all the required assertions (an invariant for each loop; an entry, and an exit assertion for each procedure; an entry assertion for each call statement) are given. The process of verification is not interactive. The student is informed only of the outcome of the verification. If his program is not proven correct, the lemmas which were false are indicated. He may then request a counterexample, or proceed directly to edit his program.

We emphasize that when a program is not proven correct, it may be because strong enough assertions were not given.

5.3.6 Possible Extensions of SORTLAB

It seems possible to construct a "sorting expert" consisting of such components as loop invariant generator, termination prover, efficiency analyzer, elegance judger, and algorithms expert. Systems similar in intent to these subcomponents have been designed in other contexts. Elspas [1973] describes how the efficiency of a program analyzed automatically, a by-product being termination. Considerable literature (see, e.g. [Wegbreit 1974]) has appeared on the automatic generation of loop invariants. Ruth [1974] discusses a system which attempts to give quality feedback to the student using built-in knowledge about specific sorting algorithms like bubble sort algorithm. An elegance judger may be readily constructed if that elusive characteristic, "elegance," of a program is quantified in terms of measurable quantities like the length of the proofs of correctness, number of statements, variables etc.

The tutoring system SORTLAB would certainly be more attractive with such a sorting expert. The construction of this component seems doable, but is another project of same magnitude as the verifier.

6. DISCUSSION

Many verifiers have been constructed. Yet, none of them can be considered a tool usable by ordinary programmers. The number and variety of programs proven is small. Data structures more complex than linear arrays or lists are handled unnaturally. More significant is their lack of performance of these verifiers in terms of memory space, and computation time needed.

This failure in making significant advances toward constructing verifiers that are mechanical aids to program writing can be largely attributed to the very attitude taken in building several of the present day verifiers. They all seem to start with the presumption: Given an arbitrary program with assertions, prove it. Evidence is building up that practically usable verifiers cannot be constructed unless the problem domain is limited, programs are well-composed, abstract data structures and operations are used, and properties of programs and data structures are studied from a semantic viewpoint. Thus, we foresee not one ultimate program verifier but a class of limited domain program verifiers, each capable of proving/disproving a certain class of programs.

Section 6.1 elaborates these points. Section 6.2 describes a few of the significant verifiers and theorem provers built so far.

6.1 A Critique of Program Verifiers

McCarthy [1963] was one of the earliest to recognize the need to replace debugging of systems (computer programs, engineering systems, etc.) by proofs that systems meet their specifications. Considering

programs as mathematical objects, he goes on to show how statements about programs may be proven. The theory developed by Floyd [1967] for iterative programs is comprehensive and equates the correctness of the program to the truthhood of a certain set of lemmas generated from it.

King [1969] constructed a verifier which mechanized both lemma generation and proof. This clearly demonstrated the feasibility of an automatic program verifier and became the pilot system for a dozen or so systems to follow (see [London 1972]). Many of these verifiers are the result of unfortunate marriages between a lemma generator and a classic automatic theorem prover, and none can be considered to be significantly superior to King's verifier.

6.1.1 Theorem Provers for Program Verifiers

Work on classic theorem proving always concerned itself with the general problem of syntactically deducing that a given statement of first-order logic follows from a set of axions (see, e.g., [Chang and Lee 1974], and [Bledsoe 1975]). Pointing out some of the theoretical impediments to automatic theorem proving, Rabin [1974] comments that this work had such high hopes and aims as:

> . . .to develop a theorem prover which will enable them to solve mathematical problems, and hopefully even difficult mathematical problems, by the computer. If one wants to slide into the realm of science fiction then one may talk about proving or disproving Fermat's conjecture by an automated theorem proving program. . . .

Since first-order logic is undecidable, one is looking only for efficient semi-decision procedures which will produce proofs of statements which

are theorems and halt, and which may not halt on nontheorems. But, as Rabin makes it plain, even in such theoretically decidable domains as Pressburger Arithmetic (first-order sentences involving natural numbers and the operation of addition only), to computationally determine if a given sentence is true or false may be practically undecidable.

If verification is ever to replace debugging, verifiers should be able to handle incorrect programs. That is, we need theorem provers which are decision procedures for the lemmas generated. Thus, the programs that a verifier attempts to prove or disprove should be so limited that the lemmas generated belong to a decidable domain. This can be done only by carefully designing a language for assertions expressive enough to allow all "legitimate" assertions one might want to make in proving properties of programs from an interesting class of programs. The theorem prover should then be a decision procedure for all sentences in the assertion language.

Since even decision procedures may take impractically long to decide if a sentence is true or false, they should be so engineered that for a large subset of the lemmas that can be considered to be "naturally occurring" in well-designed programs such decisions are made rapidly. Thus, we may not mind if it takes super-exponential time to decide if a verification condition of the following kind

$$\{\ \Omega \mid$$
$$\quad i \leftarrow i$$
$$\quad \mid\ \omega\ \}$$

is correct (because the programmer has the bad manners of misusing the verifier to prove an irrelevant mathematical theorem that Ω implies ω) so long as the verifier gives correctness proofs of legitimate programs quickly.

Furthermore, the lemmas generated in proving well-designed, legitimate programs are not typical of manual mathematics. These lemmas are shallow and follow fairly directly from (properly chosen) axioms and inference rules. Clearly, it is impractical to include all lemmas to be proven as the set of inference rules; a small number of inference rules should be carefully tailored so that short proofs of naturally occurring lemmas can be given rapidly. Two examples of theorem provers so designed are [King and Floyd 1972] and the theorem prover described in Chapter 3 of this thesis.

6.1.2 Effect of Program Composition

The structure and statements of a program clearly will have an effect on its verification. Writing abstract programs using abstract data structures has been advocated by such authors as Dijkstra and Hoare. The solution to a programming problem is constructed using operations on data structures that are natural to the problem. These operations and data structures will then be written at a lower level of abstraction, and so on, until all operations and abstract data structures are implemented in the host programming language. The advantages of such an approach lie in the factorization of detail at any given level of abstraction.

Such abstraction is helpful not only to the human designer of the program, but also to the program verifier. When data structures are manipulated solely through designated procedures, properties related to data integrity can be proven by considering these procedures independently of their invocations using generator induction [Hoare 1972]. Thus, for example, that a sorting algorithm has only permuted the given ordered set of elements can be shown by proving that the primitive operations exchange and insert were element-conserving.

Another important advantage to be gained is that undecidable domains of lemmas may be isolated in a program. Arithmetic operations such as multiplication, division and addition which result in theoretically or practically undecidable domains can be grouped together and their input/output relationships explicitly given. These relationships may then be proven separately by ad hoc techniques. Often, such arithmetic is not essential to the property of the program being proven. For example, the division by 2 in binary search, and multiplication by 2 in siftup of heap sort are not essential to the correctness proofs. The only thing that matters for the correctness of the search is that the interval of uncertainty be partitioned into two smaller subintervals.

These operations on data structures are generally implemented as procedures. Only selected components of a data structure are modified by the procedures, keeping the remaining environment of the procedure intact. However, the rules of inference about procedure calls such as those given in [Hoare 1973] or in [Elspas et al. 1973] deal only with "entire variables" (a whole array, a whole stack, etc.) and are weaker

than they should be. That is, correct programs exist which cannot be proven using such inference rules. A "predicate transformer" (a la [Dijkstra 1976]) offers a solution to this problem.

The rule of procedure invocation of [Hoare 1973] can be roughly described as follows:

> Let Q be a procedure whose correctness with respect to ϕ and ψ has been established independently, i.e.,
>
> $$\{\phi \mid Q \mid \psi\}$$
>
> Then to prove $\{\alpha \mid \underline{\text{call}}\ Q \mid \beta\}$ verify the following:
>
> $$\alpha \models \phi'$$
>
> and
>
> $$\psi' \models \beta$$
>
> where ϕ' and ψ' are obtained from ϕ, and ψ with appropriate substitutions made for the formal parameters of Q.

Clearly, this rule is sufficient to prove $\{\alpha|\underline{\text{call}}\ Q|\beta\}$. But the exit assertion ψ of Q cannot, in general, contain enough information to imply β when Q is called under different input environments, all of them satisfying ϕ'. A number of properties guaranteed by α may be unchanged by Q, and hence true upon exiting Q. What is needed is a meta-operator which produces a β' as the transformations made by Q on α when α implies ϕ'. Such an operator in the context of backward substitution is a "predicate transformer," transforming the given exit assertion β of the call Q into α', which is the weakest entry condition to call Q such that β is true if and when call Q returns.

The verifier should be given a predicate transformer for each procedure Q which may be invoked under varying circumstances. However, if the procedure Q is not well-written (e.g., global variables were used where local variables should have been used), the predicate transformer will be an overspecification of Q. It should also be realized that some procedures are called only in certain contexts. In such cases, Hoare's rule is simpler to use.

6.1.3 Proving Certain Properties of Programs

It is not difficult to invent innocent-looking programs whose correctness is very difficult to establish. Pure and deep mathematical results may be used in the program and hence there may not be a "directly perceivable" relation between what is being computed and the stated intentions of the program.

For example, a depth-first search algorithm [Tarjan 1972] computes certain simple functions NUMBER(·) and LOWPT(·) on vertices, and deletes all edges from a stack until a certain condition on NUMBER(·) is satisfied. This property is quite obvious to prove. That this set of edges constitutes a biconnected component of the graph, however, is a difficult theorem. It is interesting to note that this and several other graph algorithms use very simple arithmetic (successor function +1, and $\leq$ relation). Habermann [1975] gives another example of an algorithm (a quadratic-hash algorithm) whose correctness proof does not readily follow from the program structure itself.

"Existential" properties are also quite difficult to prove using the inductive assertion approach. Consider, for example, an algorithm enumerating all circuits of a graph. Its exit assertion is:

> Every subgraph g (of the given graph G) that is output is a circuit of G, and conversely, every circuit of G is output.

As another example, consider a shortest path algorithm. The exit assertion is:

> The graph G has no path shorter than the one found by the algorithm.

The path p found by the algorithm often appears explicitly in the algorithm, while the set of all paths of G that p is being compared to does not.

6.2 Previous Work Related to This Thesis

In a survey, London [1972] reports that there are more than a dozen verifiers constructed so far, most of these using the inductive assertion method. None of these verifiers can, in general, handle incorrect programs. Only algorithms that were known to be correct a priori have been mechanically verified with varying degrees of human intervention in their proofs.

We briefly describe two of these verifiers--King's and SRI--which have influenced the verifier presented in this thesis. Other

significant verifiers include [Luckham et al. 1973],[Deutsch 1973], [Boyer and Moore 1975], [Good et al. 1975] and [Marmier 1975]. Cooper [1975] discusses independently some ideas similar to those expressed in Chapter 3.

6.2.1 King's Verifier

King [1969] constructed a verifier which mechanized both the lemma generation, and their proof. A commendable engineering approach was taken in tailoring the theorem prover. The programs, and hence the lemmas, were limited to integer-valued variables, including linear arrays. Several ad hoc techniques which depend on the detailed knowledge of integer expressions are used in proving a large class of lemmas about integers. The premise and the negation of the conclusion of the lemma to be proven are represented in a "normal" form, and the resulting set of linear inequalities, and nonlinear equations is algebraically solved [King and Floyd 1972].

Among the programs that King's verifier has proven, without any human intervention, are: simple insertion sort, bubble sort, and computing x^y using the binary representation of y.

Subsequent verifiers ([Elspas et al. 1973], [Luckham et al. 1973], [Good et al. 1975], [Deutsch 1973]) have provided for interaction with the user in attempt to prove a much larger class of programs, resulting in the proofs of such programs as Hoare's FIND.

6.2.2 SRI Verifier

The theorem prover [Elspas et al. 1973] is a collection of inference rules together with a set of strategies. Given the premise of a verification condition to be proven, determining whether it implies the conclusion proceeds in a goal-driven manner. The theorem prover has several high-level inference rules about arrays. Unfortunately, the theorem prover is embedded in a disastrously general QA4 system [Rulifson 1972], and lacks a sense of direction. At any given point, several inference rules are applicable, and the system applies each one in turn until it succeeds in proving the goal or exhausts all inference rules when, of course, the lemma is false. However, it should be noted that the application of an inference rule may generate further instances of application for another rule, and vice versa, resulting in thrashing. The user may be called upon to provide advice on such and other occasions which can then alter the course of deduction.

Both King's verifier, and the SRI verifier handle arrays unsatisfactorily, using the equivalent of access and change functions of McCarthy [1967] because array elements are considered to be of the same type as their indices, and interassignments between them are allowed.

Our own inference rules about arrays (see Chapter 3) may be considered as refinements of the rules in the SRI verifier.

6.3 Salient Features of the Sorting Program Verifier

The verifier presented in this thesis has been designed to meet specific performance requirements. It was to be usable in an

interactive computing system which imposed severe constraints on both the amount of memory and computation time that can be used (see Section 5.1). This section briefly analyzes the factors that contributed to the fast decision procedure, and notes some of its shortcomings.

6.3.1 Decidable

The verifier presented here is unique in that it is the only verifier with a decision procedure for the verification conditions of the programs it accepts to verify. It makes no pretense of being general. The syntax of the input programs has been carefully designed to reject all programs that the verifier cannot prove or disprove. It provides two basic operations, exchange and insert, to permute the elements of the array, thereby guaranteeing that the elements of the array are conserved. The assertion language is just powerful enough to express all the assertions that may be made about sorting-type algorithms. The basic predicates provided capture the notion of sequential access in sorting algorithms.

The decidability is due to such restriction of the lemmas generated, and the partitionability of the sequentially accessed array structure. This results in a canonical representation for each lemma to be proven. The rule of local implication lets us decide if a given predicate is implied by the hypothesis without any search. At no time does our theorem prover need to backtrack or consider various inference rules for their applicability.

6.3.2 Fast

The theorem prover is not only a decision procedure, but gives

these decisions rapidly for most theorems encountered in proving sorting algorithms. It should be noted that loop invariants of most algorithms (not necessarily sorting) are conjunctions of predicates. This theorem prover is specially suited to prove such theorems by natural deduction. It might appear that a large number of linear orderings of boundaries will be considered in the proof of a lemma; however, if the algorithm is well-written this is generally not the case. Such lack of information about how the boundaries are ordered is not typical of sorting algorithms.

Two factors contributing to the speed of the theorem prover are the large inferences made about array segments, without considering their individual elements, and the rule of local implication.

6.3.3 Backward Function Evaluation

The backward function evaluation, in the context provided by the ptr expressions which constrain the boundaries of array segments, considerably simplifies a given lemma. This completely eliminates the need for such pseudo-functions as access, and change of McCarthy, used in nearly all other verifiers. It is important to realize that such contextual evaluation is valid only if assignments among array indices and elements are not permitted.

6.3.4 Counterexample Generation

We consider the generation of counterexamples one of the most important duties of a program verifier. If debugging is ever to be replaced by verification, incorrect programs must be handled by verifiers

by either suggesting corrective actions, indicating the unproven verification condition, or actually generating a counterexample for the skeptic.

As shown in Chapter 3, a modified shortest-path algorithm is the counterexample generator used by this verifier.

6.3.5 Some Shortcomings

It is interesting to note that the theorem prover is not goal oriented. Thus, in proving even a trivial theorem such as

$$\underline{\text{sorted}}\ (1,n) \models \underline{\text{sorted}}\ (1,n)$$

it considers two partitions (one for each of the cases $n \leq 0$ and $n > 0$) of the array. This is typical of decision procedures in that they may ignore shortcuts. However, the strength of our decision procedure is in its orientation toward naturally occurring theorems.

More seriously, it is hard to generalize the theorem prover. For example, if we permit the predicate that all keys of an array segment are distinct, the theorem prover cannot be extended in a straightforward manner.

6.4 Conclusion

SORTLAB shows that verifiers for programs from a limited domain of application, which incorporate some of the semantics of the domain, are practical. It would be interesting to see an approach similar to that described in this thesis tried for another domain that is well-understood and easily formalized mathematically.

We believe that such limited program verifiers will be the trend of the future, in the wake of recent results in practical undecidability and the lack of progress in mechanical program verification in general.

REFERENCES

[Bledsoe 1975]
W. W. Bledsoe, "Non Resolution Theorem Proving," ATP-29, Departments of Mathematics and Computer Sciences, University of Texas, Austin, Texas 78712, September 1975.

[Boyer and Moore 1975]
R. S. Boyer and J. S. Moore, "Proving Theorems about LISP functions," Journal of ACM 22 (1975), 129-144.

[Chang and Lee 1973]
Chin-Lian Chang and Richard Char-Tung Lee, "Symbolic Logic and Mechanical Theorem Proving," Academic Press, New York, 1973.

[Cooper 1975]
D. C. Cooper, "Proofs about Programs with One-Dimensional Assays," Unpublished manuscript, March 1975.

[Dahl et al. 1972]
O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, "Structured Programming," Academic Press, New York, 1972.

[Danielson 1975]
Ronald L. Danielson, "PATTIE: An Automated Tutor for Top-Down Programming," Ph.D. Thesis, University of Illinois, Urbana, Illinois 61801, October 1975.

[Deutsch 1973]
L. Peter Deutsch, "An Interactive Program Verifier," Ph.D. Thesis, University of California, Berkeley, California, May 1973.

[Dijkstra 1976]
Edsger W. Dijkstra, "A Discipline of Programming," Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Eland 1975]
Dave R. Eland, "An Information and Advising System for an Automated Introductory Computer Science Course," Ph.D. Thesis, University of Illinois, Urbana, Illinois 61801, June 1975.

[Floyd 1964]
Robert W. Floyd, "Algorithm 245: Treesort 3," Communications of ACM 7 (1964), 701-701.

[Floyd 1967]
Robert W. Floyd, "Assigning Meanings of Programs," Proceedings of a Symposium on Applied Mathematics, American Mathematical Society 19 (1967), 19-32.

[Elspas et al. 1973]
Bernard Elspas, Karl N. Levitt and Richard J. Waldinger, "An Interactive System for the Verification of Computer Programs," Standford Research Institute, SRI Project 1891, Menlo Park, CA 94025, September 1973.

[Good et al. 1975]
Donald I. Good, Ralph L. London and W. W. Bledsoe, "An Interactive Program Verification System," IEEE Transactions on Software Engineering 1 (1975), 59-67.

[Habermann 1975]
A. N. Habermann, "The Correctness Proof of a Quadratic-Hash Algorithm," Department of Computer Science, Carnegie-Mellon University, Pittsburg, PA 15213, March 1975.

[Hansen 1971]
Wilfred J. Hansen, "Creation of Hierarchic Text with A Computer Display," ANL-7818, Argonne National Laboratory, June 1971.

[Hoare 1971a]
C. A. R. Hoare, "Proof of a Program: FIND," Communications of ACM 14 (1971), 39-45.

[Hoare 1971b]
C. A. R. Hoare, "Procedures and Parameters: An Axiomatic Approach," Proceedings of Symposium of the Semantics of Algorithmic Languages, Lecture Notes in Mathematics 188, Springer Verlag, 1971.

[Hoare 1972]
C. A. R. Hoare, "Proof of Correctness of Data Representations," Acta Informatica 1 (1972), 271-281.

[Luckham et al. 1973]
David C. Luckham, Friedrich W. vonHenke, Shigerie Igarashi, Ralph L. London and Norihisa Suzuki, "Automatic Program Verification," STAN-CS-(73-365, 74-473, 74-475, 75-522), Standford University, Standord, California, 1973.

[King 1969]
James C. King, "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon University, National Technical Information Service, Springfield, Virginia 22151, #AD 699248, September 1969.

[King and Floyd 1972]
James C. King and Robert W. Floyd, "An Interpretation-Oriented Theorem Prover over Integers," Journal of Computer and System Sciences, 6 (1972), 305-323.

[London, R. L. 1970]
Ralph L. London, "Certification of Algorithm 245: Treesort 3," Communications of ACM 13 (1970), 371-373.

[London 1972]
Ralph L. London, "The Current State of Proving Programs Correct," Proceedings of Twenty-fifth Annual ACM Conference, 1972, 39-43.

[Manna and Pneuli 1974]
Zohar Manna and Amir Pneuli, "Axiomatic Approach to Total Correctness of Programs," Acta Informatica 3 (1974), 243-263.

[Marmier 1975]
Edouard Marmier, "Automatic Verification of PASCAL Programs," Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, 1975.

[McCarthy 1960]
John McCarthy, "Recursive Functions of Symbolic Expressions and their Computation by Machine," Communications of ACM 3 (1960), 184-195.

[McCarthy 1963]
John McCarthy, "A Basis for a Mathematical Theory of Computation," in Computer Programming and Formal Systems, P. Braffort and D. Hirschberg (editors), North-Holland 1963, 33-70.

[McCarthy and Pointer 1967]
John McCarthy and J. A. Pointer, "Correctness of a Compiler for Arithmetic Expressions," Proceedings of a Symposium on Applied Mathematics, American Mathematical Society 19 (1967) 33-41.

[Misra 1976]
Jayadev Misra, Private Communication, 1976.

[Naur 1966]
Peter Naur, "Proof of Algorithms by General Snapshots," BIT 6 (1966), 310-316.

[Nievergelt 1975]
Jurg Nievergelt, "Interactive Systems for Education - the new look of CAI," Proceedings of IFIP World Conference on Computers in Education, North Holland 1975, 465-471.

[Rabin 1974]
Michael O. Rabin, "Theoretical Impediments to Artificial Intelligence," Information Processing, 1974, 615-619.

[Reingold 1973]
Edward M. Reingold, "A Nonrecursive List Moving Algorithm," Communications of ACM 16 (1973), 305-307.

[Rulifson et al. 1972]
J. F. Rulifson, J. A. Derksen and R. J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning," Final Report, SRI Project 8721, Standford Research Institute, Menlo Park, California, 1972.

[Ruth 1974]
Gregory R. Ruth, "Intelligent Program Analysis," Massachusetts Institute of Technology, Cambridge, Massachusetts, Manuscript, February 1974.

[Sherwood 1975]
Bruce A. Sherwood, "The TUTOR language," Computer-Based Education Research Laboratory and Department of Physics, University of Illinois, 1975.

[Tarjan 1972]
Robert E. Tarjan, "Depth First Search and Linear Graph Algorithms," SIAM Journal on Computing 1 (1972), 146-160.

[Wagner 1974]
Robert A. Wagner, "A Simple List Moving Algorithm," Vanderbilt University, Nashville, Tennessee, Manuscript, 1974.

[Wegbreit 1974]
Ben Wegbreit, "The Synthesis of Loop Predicates," Communications of ACM 17 (1974), 102-112.

[Wilcox et al. 1976]
Thomas R. Wilcox, Alaine Davis and Michael Tindall, "The Design and Implementation of a Table Driven, Interactive Diagnostic Programming System," Communications of ACM 19 (1976), to appear.

APPENDIX

Performance of the Verifier - An Example

The following selection sort program has a weak assertion at 7*.

```
1   procedure  sort (n)
*   TRUE
2      scan up with i from 1 to n-1
3          scan up with j from i+1 to n
4             if xi > xj then
5                exchange xi with xj
6             else
7             endif
*             S(1,I) ≤ XI ≤ A(I+1,J) & 1 ≤ I < J ≤ N
8          endscan
*          S(1,I) ≤ A(I+1,N) & 1 ≤ I < N
9      endscan
*   S(1,N)
10  endproc
```

The theorem prover disproves the corresponding verification condition.

(subst j-1 for j in 7*) and j ≤ n

stnts [4...7] b (7*)

in 1114 CPU-milliseconds. When the assertion at 7* is given as:

S(1,I-1) ≤ A(I,N) & XI ≤ A(I+1,J) & 1 ≤ I < J ≤ N

the program is proven correct in 9346 CPU-milliseconds.

VITA

Prabhaker Mateti was born in Mahbubabad, Andhra Pradesh, India, on June 18, 1948. He graduated from Osmania University with a B. E. in Electrical Engineering in December 1969. He received his M. Tech. in Electrical Engineering from the Indian Institute of Technology Kanpur in May 1973. He has been a research assistant in the Department of Computer Science from September 1972 to August 1975. He was an Instructor at the University of Texas at Austin from September 1975 to August 1976.

BLIOGRAPHIC DATA IEET	1. Report No. UIUCDCS-R-76-832	2.	3. Recipient's Accession No.
Title and Subtitle An Automatic Verifier for a Class of Sorting Programs			5. Report Date October 1976 6.
Author(s) Prabhaker Mateti			8. Performing Organization Rept. No. UIUCDCS-R-76-832
Performing Organization Name and Address Department of Computer Science University of Illinois at Urbana-Champaign Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. NSF EC 41511
Sponsoring Organization Name and Address National Science Foundation Washington, D.C. 20550			13. Type of Report & Period Covered Ph.D. Dissertation 14.

Supplementary Notes

Abstracts

A decision procedure for the verification conditions generated from a class of in-place sorting algorithms is presented. Counter-examples to false verification conditions can be generated. The special techniques developed seem applicable to a wider class of programs that manipulate data structures.

A programming laboratory, called SORTLAB, for beginning students has been implemented. SORTLAB consists of a program editor, recognizers for the programming and assertion languages tailored for in-place sorting, the program verifier and an interpreter.

Key Words and Document Analysis. 17a. Descriptors

SORTLAB
Theorem Prover
Program Verifiers

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release Unlimited	UNCLASSIFIED	122
	20. Security Class (This Page) UNCLASSIFIED	22. Price ----------